

大话存储

终极版

张冬 著

清华大学出版社

张冬（网名：冬瓜头）

存储行业混迹十载，历任售前售后、产品设计、咨询顾问等角色，四项存储技术专利发明人。酷爱钻研，个性执着。理想是晚年能够做一名教师。

内容简介

网络存储是一个涉及计算机硬件以及网络协议/技术、操作系统以及专业软件等各方面综合知识的领域。目前国内阐述网络存储的书籍少之又少，大部分是国外作品，对存储系统底层细节的描述不够深入，加之术语太多，初学者很难真正理解网络存储的精髓。

本书以特立独行的行文风格向读者阐述了整个网络存储系统。从硬盘到应用程序，对这条路径上的每个节点，作者都进行了阐述。书中内容涉及：计算机IO基本概念，硬盘物理结构、盘片数据结构和工作原理，七种常见RAID原理详析以及性能细节对比，虚拟磁盘、卷和文件系统原理，磁盘阵列系统，OSI模型，FC协议，众多磁盘阵列架构等。另外，本书囊括了存储领域几乎所有的新兴技术，比如机械磁盘、SSD、FC/SAS协议、HBA卡、存储控制器、集群存储系统、FC SAN、NAS、iSCSI、FCoE、快照、镜像、虚拟化、同步/异步远程复制、Thin Provision自动精简配置、VTL虚拟磁带库、数据容灾、应用容灾、业务容灾、性能优化、存储系统IO路径、云计算与云存储等。

其中每一项技术作者都进行了建模和分析，旨在帮助读者彻底理解每一种技术的原理和本质。本书结尾，作者精心总结和多年来在论坛以及各大媒体发表的帖子内容，超过一百条的问与答，这些内容都是与实际紧密结合的经验总结，颇具参考价值。

本书第一版于2008年出版，受到业界一致肯定，历经6年技术沉淀重装出版。

本书适合初入存储行业的研发人员、技术工程师、售前工程师和销售人员阅读，同时适合资深存储行业人士用以互相切磋交流提高。另

外，网络工程师、网管、服务器软硬件开发与销售人员、Web开发者、数据库开发者以及相关专业师生等也非常适合阅读本书。

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。

版权所有，侵权必究。侵权举报电话：010-62782989 13701121933

图书在版编目（CIP）数据

大话存储：终极版.存储系统底层架构原理极限剖析/张冬 著. —北京：清华大学出版社，2014

ISBN 978-7-302-38124-2

I. ①大... II. ①张... III. ①计算机网络—信息存储 IV. ①TP393

中国版本图书馆CIP数据核字（2014）第224466号

责任编辑：栾大成

封面设计：

责任校对：徐俊伟

责任印制：

出版发行：清华大学出版社

网 址： <http://www.tup.com.cn>, <http://www.wqbook.com>

地 址： 北京清华大学学研大厦A座

邮 编： 100084

社 总 机： 010-62770175

邮 购： 010-62786544

投稿与读者服务： 010-62776969， c-service@tup.tsinghua.edu.cn

质量反馈： 010-62772015， zhiliang@tup.tsinghua.edu.cn

印 刷 者：

装 订 者：

经 销： 全国新华书店

开 本： 188 mm×260 mm

印 张： 63.5

插 页： 1

字 数： 1668千字

版 次： 2015年1月第1版

印 次： 2015年1月第1次印刷

印 数： 4000

定 价： 149元

产品编号：061141-01

作者感言

各位读者好，很高兴再次为大家“大话”存储，记得上一次是在3年前。6年前，当《大话存储》一书在2008年出版面世之后，我当时就许下承诺，要写《大话存储终极版》。当时之所以敢夸下海口要继续写第二本，是因为《大话存储》介绍了存储领域最基本的概念和架构，但并没有深入涉及存储领域最新的技术，比如重复数据删除、Thin Provision、动态分级存储、CDP连续数据保护、SSD固态硬盘、FCoE、SAS、云计算和云存储等。

当年的《大话存储》确实满足了广大读者的一定需求，出版之后也获得了诸多好评和官方的、民间的很多奖项。这些成果逐渐让我感觉到更大的责任和压力。正因如此，所以我深知绝对不能就此停歇，学习是永无止境的，技术是不断发展的，所以我先向大家做了承诺，这样就可以无时无刻地激励我继续学习研究下去了。

写作过程是极其困难的，尤其是当一字一句都需要精雕细琢，并且时刻以通俗表达且让所有人都能看懂的原则和基准去写的时候，其所耗费的精力和脑力是巨大的。记得在一年前撰写本书主体的时候，基本上每天都是早晨七八点钟起来，从床上直接到书桌前开始写，直到中午吃饭，吃饭过程中依然在脑海中构思着，就这样一直到晚上，最晚的一次记得是做一个实验，通宵达旦，直到第二天天亮，实在体力不支，去床上躺到中午，然后继续写。每次睡觉之前，都会带着一个疑问入睡，躺下之后就在脑海中构思、建模，一旦想到某些重要的东西，就用笔记下几个关键词，否则第二天准忘。大部分情况一般都是没想到什么思路就已经呼呼大睡了。这种状态持续了半年之久，当完成了主体稿件之后，

真的有一种如释重负的感觉。可惜，好景不长，随着不断的学习和深入，逐渐发现已经写完的内容当中有大量需要补充完善、修饰的部分，在修饰完善的过程中，继续思考，结果发现又引申出更多的东西，有些甚至推翻了以前的结论。这种状态又持续了半年，最终定稿交给编辑之后，依然发现还有零碎的东西需要完善甚至推翻，结果一再将更新的内容同步给编辑，导致出版日期一推再推，出版社相关编辑、校对叫苦不迭，还好咱的老战友大成编辑一如既往地支持，我们都顶住了压力，直到最后一个月时间内没有再发现需要完善的内容，达到了最终收敛。后面这个过程感觉更加耗费精力，因为当你重新审视之前内容的时候，一旦发现不完善甚至错误，就会感觉到一种挫败感和愧疚感，使你的激情和斗志有所丧失。

写书不但是给他人共享知识的过程，它更是一个总结自身知识体系、提高自身修养以及让自己学习更多知识的途径。比如，我在写书过程中，不但通过各方面渠道纠正了之前对某项技术的一些错误认识，而且还学习了更多的知识，并且将这些知识进行深度理解分析，之后通俗地表达出来。当你发现其他人通过你的知识快速提高之后，这种感觉是最充实的。只有在奉献之后才会感到充实，而不是一味的去索取，这样只能更加空虚。

本次终极修订版，一是针对《大话存储》和《大话存储终极版》这两本书中的错误进行修改，二是将前两本书彻底整合成为一本，三是增加了一些前沿内容，四是针对前作中读者反映不太好懂的地方进行了重新诠释。鄙人的下一本书已经在写作当中，本着十年磨一剑、磨不好绝不拿出来的态度，相信不会让读者失望。

感谢家人对我的支持！长达半年的无业状态，没有家人支持就没有这本书。

感谢那些曾经帮助过我的不计其数的网友和同事！没有鼓励也不会有这本书。

感谢清华大学出版社的工作人员为本书所付出的工作！没有信任更不会有这本书。

感谢华中科技大学武汉光电国家重点实验室博士生导师谢长生教授对鄙人和本书的大力支持！

感谢本书的广大读者，你们的支持给了我持续前进的动力！

致本书最终修订版交稿之前，我的女儿诞生了，感谢妻子。不知道她成人之后，能否还能看到20多年前的这本由他父亲所著的著作，虽然那时候本书可能早已过时。我会努力把钻研、毅力、执着、忘我这些亘古不变的东西，在女儿成长过程中传授给她，我相信，有了这些，做什么都会成功。

作者联系方式：

QQ: 122567712

Email: 122567712@qq.com; myprotein@sina.com

MSN: myprotein0007@hotmail.com

新浪微博: @传说中的冬瓜头

Blog: <http://space.doit.com.cn/35700>

读者交流QQ群: 72168388、361934810

序1

我关注张冬这个名字是在《大话存储》一书刚出版的时候。作为一个长期从事信息存储技术与教学的大学教师，自认为对于国内外关于网络存储方面的各种书籍和资料比较熟悉，对业界有哪些高人也算比较了解，但我在书店偶然发现一本名为《大话存储》书的时候，确实感到有点意外和惊喜。好像在熟悉的武林圈子之外，突然出现一位武林高手在那里论道。好奇心驱使我赶紧买了一本书回家研读，结果发现这本书确实与众不同。

与我们这些所谓学院派写的中规中矩的书相比，此书风格特立独行，语言形象生动，潇潇洒洒，颇具武侠之风。书中充满着智慧的思考和有趣的比喻，将各种原本枯燥深奥的技术概念和原理论述得十分透彻明白。不仅如此，该书还收集了大量的实例，使读者在系统获得网络存储知识的同时，还能了解典型实际系统的工作原理和技术细节，具有很好的实用性。我读完之后，对这本书的作者十分好奇。一个80后而且还是学化学出身的年轻人，如何就能写出这种行文老到而风格独特的专业技术书籍呢？上网查了一下冬瓜头（张冬的网名）的技术博客和他在各种论坛留下的文字，我得到了答案。这是一个完全由兴趣驱动而对技术极端痴迷的人，也是一位善于思考、富于想象力的人。这种纯粹的、不含任何功利成份的兴趣与痴迷，才是促进科学技术发展的真正源动力。

真正和张冬接触，是因为他来信质疑我们实验室申报的一项专利。收到质疑的来信，我和提出这项专利的博士生经过仔细研究，发现我们提供的图上因为少了一个非门，结果将会因为反相而出错。对如此细致具体的问题，一般人是难以发现的。如果没有打破砂锅问到底的较真精

神，哪里会发现如此细节的错误呢？这种质疑的精神，在科学研究中是极为宝贵的。我们学校被称为“根叔”的李培根校长，在2010年的新生开学典礼大会上，就以“质疑”为题作了讲演，激励青年学子发扬质疑精神。有质疑精神的人，不唯上，不唯权威，只认真理，这正是我们这个时代所稀缺的精神。

强烈的兴趣，对技术的痴迷，加上质疑精神，成就了一本存储领域的一本好书。我在研究生新生入学之后，就推荐他们先读一下《大话存储》这本书。一方面此书对研究生而言，确实是一本网络存储技术入门的好书，另一方面我还有一个用意，就是让他们知道，要从事科学研究，强烈的兴趣比什么都重要。

信息存储是信息跨越时间的传递，也是人类传承知识的主要手段。在信息存储技术上，人类有超过万年的发明创造史。早期就地取材，人类利用石刻、泥板、竹简和羊皮等来记录信息，后来发明了纸张和活字印刷来保存和传播信息，近代发明了照相、录音和录像技术来存储信息。利用这些发明和创造，人类留下了极为丰富的文字、绘画、图像、语音和视频信息。正是这些信息，记录了人类创造的知识体系，使我们能够传承文明，并在此基础上创造新的文明。

从计算机的发明为开端，人类的信息技术进入了一个以数字化为特征的历史性新阶段。各种形式的信息被转换成数字后，以统一的方式进行处理、传输和存储，然后再转换为各种形式的信息被人们所利用。这种前所未有的方式发明之后，一个以数字化为特征的信息革命浪潮就波澜壮阔地形成。各种信息都被大规模数字化，使数字化的信息呈爆炸性增长。特别是互联网的兴起和普及，大大加快了信息的流通过程，使数字信息加速产生。图灵奖获得者Jim Gary观察这种数据急速增长的趋势后，总结出一个规律：人类每18个月新增的数据量，将是历史上所有数

据量之和！如此下去，对信息存储的需求将是无止境的，信息存储技术在这种强烈的需求驱动下得到了空前的发展。

为了保存数字化的信息，当代的科学家和工程师在最近的几十年中发明了磁存储、光存储、半导体存储等多种存储技术，其中大容量的硬盘在海量信息存储中扮演了主要的角色。硬盘的密度在短短几十年中增长了一百万倍以上，在近期，硬盘密度每年增长都接近一倍，而且还有不小的增长空间。由硬盘作为基本单元，通过各种总线、网络将硬盘连接成不同层次和不同规模的存储系统，就构成了我们目前的网络存储系统。例如由硬盘组加上冗余纠错技术构成磁盘阵列，再由磁盘阵列通过局部高速网络连接形成存储区域网；又如通过包含硬盘的大规模集群和文件系统形成的海量存储系统成为大型网站和数据中心新的存储架构。人们发明了各种技术来提高存储系统的容量、性能、效率、可用性、安全性和可管理性。存储虚拟化、归档存储、集群存储、云存储、绿色存储等新名词不断涌现，SSD固态存储、重复数据删除、连续数据保护、数据备份与容灾、数据生命周期管理等新技术层出不穷，令人应接不暇。

在这种情况下，广大的信息领域的从业人员，信息系统的用户，以及学习信息技术的大学生和研究生，迫切需要一本既全面论述网络存储技术原理，又有丰富实例；既反映最新技术进展，又通俗易懂的书来满足他们的需求。冬瓜头的《大话存储》就是这样一本恰逢其时的好书。

《大话存储》已在业界产生了很大的影响，对存储技术在我国普及起到了良好的推动作用。该书还被引进到我国的宝岛台湾，可见其影响深远。张冬再接再厉，以他对技术的痴迷继续钻研，对第一本书作了工作量巨大的改动与增补，并增加了云存储等全新的三章内容，全面反映了他对技术的重新思考和对最新技术的深刻理解。我相信，这些新的

内容将给读者带来惊喜。

在技术发展十分迅速的领域，赶时髦的书籍多如牛毛，书店里充满了应景之作，真正经过深入思考、用心写作的书是不多的。而《大话存储终极版》却是一位技术高手的呕心沥血之作，书中对每一项技术的介绍都经过深入的思考和反复的推敲，这在当前浮躁的气氛中显得弥足珍贵。在《大话存储终极版》即将出版之际，我要向作者表示深深的敬意和衷心的祝贺，并郑重向读者推荐这本学习网络存储技术的好书。

华中科技大学计算机学院 教授

信息存储系统教育部重点实验室 主任

谢长生

序2

第一次听说冬瓜头是因为《大话存储》，而见到冬瓜头本人时，已经听他说在写《大话存储》第二版了，而今天则是欣然为《大话存储终极版》作序。我们一起有过不长时间的沟通，全是讨论最新的存储技术。有一些内容，我相信他在书中都有写到，在同他见面前，我一直在嘀咕怎么去跟他沟通。但是，看到现实中的他朴实、憨厚、腼腆，但是对技术极其敏感，说起来一套一套的，我就开始被这个山东大汉折服了，还好我是做了准备的，否则非被问倒不可。

他是个靠笔说话和表达的人，在网络论坛里，写的文字常常是洋洋洒洒，时而言辞激烈，时而意气风发，这分明是一位才高八斗的江南才子，又像是一个书场中幽默诙谐的说书人。从此之后，我经常关注他的个人博客。他会经常在博客释放一些思想出来，豪放不羁，甚至还写过长诗以及各种各样的打油诗，有些还写得非常棒，配上那个流着鼻涕的冬瓜头漫画形象，真是绝配了。

这么一个内秀的北方大汉，用豪放的气势描述一个个生涩、枯燥的技术领域，他的思想遍布他的文字，通过笔端流放在读者面前，并且还一直这么坚持。听他说，终极版出来后，还会继续写存储领域的一些细分技术，毕竟技术日新月异，尤其是在IT领域。正如他所言，昨天的中国同仁在存储技术还是个初学者，今天已经开始从蹒跚学步到自主创新了，而明天，有什么理由不能期盼他们引领潮流的身影呢。我想，这也是冬瓜头要写书的动力所在吧。

《大话存储终极版》的初稿篇幅已经超过了1500页。在浏览了全部

章节之后，发现这1500页中真的是字字珠玑！看得出来，是冬瓜头一个字一个字写出来的。更加可贵的是，全书字里行间透着他那独特的思想，对技术、对世界的理解以及他做人的态度。能够将这些世界观的东西融入一本技术书籍，这在以前是绝无仅有的！比如书中多次提到“轮回”、“阴阳”等，最后还有一节是用中医的思想来“诊治”系统性能瓶颈，看后真是令我等感叹至极！世间万物都是相互联系的，都可以找到类比和轮回，这也是冬瓜头所描述的世界观的一种。

在和冬瓜头的交谈中获知，他大学学习的专业是化学，因为高中时他化学成绩最好，所以就报了化学专业，而且期间还自学过分子生物学领域的内容，我更加惊讶了！按照他的话来说，就是“兴趣是第一驱动力”。是的，好奇和探索正是人类不断发展的第一动力。说到这里我对《大话存储终极版》中关于冬瓜头所设想的“机器如何认知自身”这段内容产生了强烈共鸣，人可以认识自身认识世界，那么机器为何不能呢？强烈的好奇心可以创造奇迹！可以让机器开口，可以让机器进化！这也正是冬瓜头所表述的世界观的一种！

《大话存储终极版》对各项存储技术的细节描述已经可以说是达到了研发级别，有很多部分甚至可以指导我们的研发！但是他却并没有用代码来表述，而是用通俗的语言和详实的图示，将原本通过阅读代码才可以理解透彻的原理，就这么轻而易举地表述了出来，这是目前我所看到的任何存储书籍或者文章都没有做到的。就这一点我曾经问过冬瓜头，问他如何做到的。他每次的回答都很简单，一针见血，实实在在，他说：“因为我就是一个从不懂钻到懂的草根，我深知一个根本不懂存储的人最想了解的东西和切入角度，并且愿意毫无保留地帮助其他草根生长！”是啊，只有亲历过那悬梁刺股的学习之路的不易，才能产出精华！

在与冬瓜头的交谈中，他还常提到一句口号：“振兴民族科技。”从他说话的眼神和口气看得出来，振兴民族科技已经成为他的信仰。他也说到，他现在所做的一切都围绕着这个信仰，他愿意为中国存储事业鞠躬尽瘁、死而后已，出版《大话存储》只是他要做的第一个环节而已，今后他还会有一系列的动作来兑现他的诺言。信仰可以改变一个人的心态与行为，我们目前太缺乏信仰，我想如果我们所有人都有这种信仰，那么“振兴民族科技”这句口号早就可以实现了。

信息存储已经成为了一个时刻影响人们生产、生活的新兴产业，它的发展也代表着世界未来的发展，让我们再来看看国产存储信息产业的发展正在经历着怎样的变革和转变。

我国处在“十二五”时期，“十二五”期间我国要实现三大转变目标：从国强到民富、从外需到内需、从高碳到低碳。这也意味着国家的发展需要依靠科技，需要大力发展新技术，尤其以信息化技术为主轴，信息化技术的发展带动重点工程的进行，势必对国产产品催生更大的需求，我相信存储业也会有更多的民族产业佼佼者诞生。

IT环境日益复杂，数据量快速膨胀，存储业也进入了一个技术更新极为活跃的黄金发展时期，产业发展迅速，技术活跃度高，这对国内厂家来说，无疑是一个脱颖而出的良好契机。那么，我们如何在这个时代背景下产生代表着民族存储业的国产佼佼者？

在这个时代，我们应该遵守什么？我们应该坚持什么？商业道德、创新精神、客户意识，我想只有将这些融入到企业性格中才能为企业注入新的活力。作为一家有理想的企业，需要具备一定的时代精神，而在存储技术日新月异的今天，企业打造独有的技术张性，终究才会超越历史，才会产生新时代的民族企业。我相信现在越来越多的企业正朝这个

方向发展。

《大话存储》以通俗易懂的语言、风趣的行文手法向读者阐述枯燥难懂的技术精髓，致力于存储信息技术发展的民族企业也同样可以在深刻理解本土文化精髓的前提下为中国写下辉煌的历史篇章。我想这个世界没有什么不可能的，只要有这份热情、专注和执著，又有什么是不可能实现的呢？

这样的一本特立独行的书，它就是时代的产物，它就是时代的精髓。

爱数软件

李基亮

序3

存储是个大市场，有意向在数据和信息系统上做投资规划的企业逐年增加，这标志着越来越多的企业意识到自身的数据安全问题。

在我十几年前刚刚踏入存储圈子之时，数据安全问题只被金融、电信等少数行业所考虑，而如今，几乎各个行业都存在数据保护与信息安全的需求。随着用户需求的急速增长，无论是硬件设备还是软件产品都是生机一片。但是，多年来我国的这个领域一直被国外产品所垄断，究其原因，是我国存储领域技术相对滞后。

我们在经营企业的过程中，花费了大量的精力进行人才的培养。在国内，计算机行业的传统教育大多集中于软件应用与网络维护上，对于专业存储的技术培训几乎为零，而存储行业又在飞速地发展着，因此，存储市场的需求与人才滞后的落差越拉越大，我们急切渴望拥有存储专业的人才去发展存储领域。“人才为本，教育当先”，人才的培养离不开教育。多年以来，存储领域的教材乃至书籍几乎是一片空白，有的也只是太过于教条以及模式化的书籍，当看到张冬先生的《大话存储》后，我深刻地体会到我国存储领域开始有了专业的教科书，我国的存储业生机盎然。

之所以赋予《大话存储》如此高的评价，是因为它的语言通俗而不失专业，幽默而不失严谨。张冬先生用读者极易接受的语言道出了存储领域的精髓。对于初学者来说，能使存储领域不再陌生，而又充满吸引。我曾了解到，《大话存储》已经成为某院校计算机专业的教材，这不仅是存储业的幸事，同时也是现代教育的幸事。坦率地讲，我们做企

业，时刻关心教育的发展，我们需要新鲜的血液来继承和发展我们的事业。《大话存储》作为能够真正做到学以致用教材之一，使我们倍感欣慰。我为我们选择的存储道路之前景充满信心，为振兴我们的民族工业充满信心，同时，为张冬这样的后继人才而倍感骄傲。

《大话存储》能够成为教材是张冬对于存储领域不懈努力的成果，《大话存储终极版》的出版，更是他不断追求与探索的结果，而《大话存储终极版》在《大话存储》的基础上更加深入地剖析了存储技术，以及存储在如今市场的广泛应用。书中不乏一些当今企业的存储实例，也包含了国内外软硬件厂家的存储技术应用，加入了更多实际范例，使读者更易理解，同时具有很强的应用性。

我相信《大话存储终极版》会给广大读者很大的帮助，同时也希望此书能够带领更多的有识青年进入存储领域，为我国民族产业的振兴而奋斗。

火星高科 总经理

龚 平

序4

认识张冬，是因他的《大话存储》，我曾在去年拜读此书，感觉一个80后的小伙子能用如此通俗的语言诠释存储技术，实属存储行业的一大喜事。这本书，可以让不了解存储的人认识存储，能够了解到存储并不是高深莫测的，即使一个存储行业以外的人去阅读《大话存储》，也一定能够读懂。用什么样的语言和叙述方式不重要，重要的是把要说的说明白。

张冬本人就像他的书一样，饱含着严谨的作风和真诚的态度，而又不乏幽默的风格。看过他的BLOG，人气一直很旺，这个致力于为国产存储业做出贡献的年轻人更是让我对他刮目相看。他在博客中写到：“我所能够做的，只有让中国人，让所有中国存储行业的人，以及中国存储行业本身，有一个扎实的基础。如果能够促进国产存储软件硬件的发展，那鄙人就是鞠躬尽瘁，死而后已，死而无憾！”一个80后年轻人有这样的雄心壮志，我们有什么理由不去努力不去发展国产存储业呢？

记得十几年前，我刚刚进入存储领域，那时候相关的书籍非常少，完全要靠自己进行反复的试验。那时（IT行业根本不成形，姑且称作计算机行业）计算机业的从业者都是抱着掌握20世纪末最具科技含量的技术的心态进行工作，从根本上说，对存储技术充满了崇拜，甚至有一丝恐惧。在探索期间，也走了不少弯路，耽误了很多时间。如果那个时候有这样一本关于存储的书籍，那简直是一大幸事！书中并没有把存储看做是多么高深的技术，而是任何一个普通人都能掌握的技术。我和张冬开玩笑说，如果你早生10年，你就可以带领我们走向一条存储道路的捷

径。

看到张冬最新力作《大话存储终极版》时，我就感觉到这又是一本好书。不仅延续了《大话存储》中通俗易懂的语言及“武侠”式的章节回目，在技术深度上，也有很深的挖掘。书中不仅囊括了时下最先进的“云”技术以及持续数据保护（CDP）技术，还涉及到了很多非常底层的架构。在《大话存储》的基础上，有了更为深刻的剖析。值得一提的是，张冬在最后还加入了Q&A的内容，把几年来读者以及网友提出的问题一一列出，并作出详细的解答，能够体会张冬在这一年多的时间里，对于存储技术的探索花了很大的心思。最可贵的是，这个年轻人不以如此成就为骄傲，继续孜孜不倦地探求。

《大话存储终极版》是一本好书，作者那严谨而真诚的态度以及致力于发展本国存储业的信心注定能够成就这样一部优秀的作品。我完全有理由相信此书能够给从业者乃至热爱存储的读者带来帮助。从中，你会受益匪浅，并乐意向你的朋友推荐此书。

火星高科 技术总监

黄 疆

关于书中的武侠情节

本书前几章存在一些武侠情节，这些情节自从本书第一版之后，收到了褒贬不一的评价，有些读者甚至看到这些情节之后非常愤怒，觉得作者在装B，从而从心理上无法接受，也就没有继续阅读。对此我深表遗憾，也正因如此，个人觉得有必要在此澄清一下这些情节的由来，以及里面那些诗句的由来。

天将降大任于斯人也，必先苦其心志，劳其筋骨，饿其体肤，空乏其身，行拂乱其所为。通俗地写出存储系统的全貌，应该也算是个大任了，但是这个大任却偏偏降给了一个对存储丝毫不懂的、甚至连计算机基本原理都不懂的人身上——老天捉弄人啊！

2005年第一次知道有SAN这个东西，2006年就被某外包公司外派到北京某大客户处担任技术顾问。读者看了会纳闷，你那时候就是一个苦B屌丝，还能当“顾问”？这一点问得好，其实我也感觉不可思议，现在想想还得感谢那个让我非常苦B的外包公司，合同是和人事代理公司签的，编制算是这个外包公司的，然后这个公司再把我派到他的客户那里，相当于倒了两次手的苦B。我从这个公司离职的时候，邮箱里面除了一封垃圾邮件，什么都没有，往事不可追，如冷风吹。

但就是这样一个公司，竟敢让我这样一个小白担任所谓“顾问”，我非常佩服他们全球排名靠前的成本控制能力，也很感激能有这个机会。当时我研究和学习计算机网络，是某个比较知名论坛的版主，而恰逢他们那个大客户在研究和模拟一种网络协议，挺复杂搞不懂，需要看看外包服务商那里有没有懂这方面的，开价估计也不菲（现在我也不知道当

时到底给我的报酬占客户开价的比例，我估计最多能有30%），所以公司内部有人推荐了我，霸王硬上弓，就这样我从家乡只身前往帝都了。项目做了一年半，我也成功地完成了任务，把协议从上到下分析地很透彻。当然，我付出的代价就是，早6点起来学习，吃完晚饭继续学习。学什么呢？计算机基本原理、通信、网络、协议，当然，还有存储。一个连8Bit=1Byte都不知道的人，学了点儿网络基本原理，就去给人当顾问，最后还成功完成了任务，听上去挺不可思议的，但就是这样发生了，笨鸟先飞，只要付出，总有回报。

《大话存储》就是在这期间写了一大半，可想而知，写作过程是痛苦的也是值得回味的，大话存储一开始其实就是个笔记，从不懂到慢慢懂，慢慢积累，慢慢提高。无时无刻不在思考，吃饭、睡觉、上厕所，满脑子都是“为什么”，然后就去求证，求证过程是痛苦的，每个材料上描述方式、结论都不一样，这直接影响了判断，这个过程很痛苦。

正是因为自己当时的苦B经历，姥姥不疼舅舅不爱，只身一人在北京待了一年半，这一年半里说的话都能数过来有多少句，再加上对知识的原发性兴趣和渴望，让我产生了某种升华，也就是所谓“侠道”，置之死地而后生，闭关苦心修炼，冥思苦想，孤独一世，独孤求败，这正是武侠的情节。我在写作过程中，不知不觉就开始自编自演，仿佛自己就是书中所描写的那些角色、那些遭遇，同时自己也希望有书中那些绝世高手的指导和提携，但实际上没有，我就是一颗无人知道的小草，我就自己提携自己，相当于“自举”，自己激励自己，从而有了那些武侠情节。退一步讲，就算不出现武侠，也可能出现另一种方式来自我激励，比如咆哮体、自嘲体、暗讽体等等，但是当时毕竟年轻，心态还是积极的，所以武侠也可能是必然的。没有这些情节激励自己，就根本无

法坚持下去，何况一个外行来写书，“大任”就得“行拂乱其所为”才能成功，大任，就得去疯狂地做、用异于常规的办法来做，才能成功！当然，不是我故意乱的，而是它必定要疯狂，必定要乱！

有人问我平时是不是个武侠迷，我的回答是：不是，我小学看过射雕英雄传但是基本就是为了看而看，因为那时候电视上播，就想找书看看，但是根本就没看懂。再往后就没看过武侠小说，武侠剧倒是看过一些，比如白眉大侠之类。这也说明了，侠道并不是刻意为之，而是由环境、经历和思考自然产生的一种境界，看武侠的不一定有侠道，能体会侠道的，不一定爱看武侠。我相信对技术有着纯粹追求和渴望的人，心中都有侠道。

看完了上述描述，我想读者如果再看到那些情节，不妨想想当年自己有多苦B，如果产生了共鸣，我想一定也会理解作者写书时候的苦B场景，这样就不至于一开始就被作者的所谓“装B”封闭了阅读下去的道路了。

最后，借书中一首诗，表达我此时此刻的感想：

七星阵里论七星，
北斗光前参北斗。
不知天上七星侠，
如今过活要饭否？

感谢各位读者，如今我已经没那么惨了，也写不出这种情节了。所谓生于忧患，死于安乐。

冬瓜头 于 北京

目 录

[作者感言](#)

[序1](#)

[序2](#)

[序3](#)

[序4](#)

[关于书中的武侠情节](#)

[第1章 混沌初开——存储系统的前世今生](#)

[1.1 存储历史](#)

[1.2 信息、数据和数据存储](#)

[1.2.1 信息](#)

[1.2.2 什么是数据](#)

[1.2.3 数据存储](#)

[1.3 用计算机来处理信息、保存数据](#)

[第2章 IO大法——走进计算机IO世界](#)

[2.1 IO的通路——总线](#)

[2.2 计算机内部通信](#)

[2.2.1 IO总线是否可以看作网络](#)

[2.2.2 CPU、内存和磁盘之间通过网络来通信](#)

[2.3 网中之网](#)

[第3章 磁盘大挪移——磁盘原理与技术详解](#)

[3.1 硬盘结构](#)

[3.1.1 盘片上的数据组织](#)

[3.1.2 硬盘控制电路简介](#)

[3.1.3 磁盘的IO单位](#)

[3.2 磁盘的通俗演绎](#)

[3.3 磁盘相关高层技术](#)

[3.3.1 磁盘中的队列技术](#)

[3.3.2 无序传输技术](#)

[3.3.3 几种可控磁头扫描方式概论](#)

[3.3.4 关于磁盘缓存](#)

[3.3.5 影响磁盘性能的因素](#)

[3.4 硬盘接口技术](#)

[3.4.1 IDE硬盘接口](#)

[3.4.2 SATA硬盘接口](#)

[3.5 SCSI硬盘接口](#)

[3.6 磁盘控制器、驱动器控制电路和磁盘控制器驱动程序](#)

[3.6.1 磁盘控制器](#)

[3.6.2 驱动器控制电路](#)

[3.6.3 磁盘控制器驱动程序](#)

[3.7 内部传输速率和外部传输速率](#)

[3.7.1 内部传输速率](#)

[3.7.2 外部传输速率](#)

[3.8 并行传输和串行传输](#)

[3.8.1 并行传输](#)

[3.8.2 串行传输](#)

[3.9 磁盘的IOPS和传输带宽（吞吐量）](#)

[3.9.1 IOPS](#)

[3.9.2 传输带宽](#)

[3.10 固态存储介质和固态硬盘](#)

[3.10.1 SSD固态硬盘的硬件组成](#)

[3.10.2 从Flash芯片读取数据的过程](#)

[3.10.3 向Flash芯片中写入数据的过程](#)

[3.10.4 Flash芯片的通病](#)

[3.10.5 NAND与NOR](#)

[3.10.6 SSD给自己开的五剂良药](#)

[3.10.7 SSD如何处理Cell损坏](#)

[3.10.8 SSD的前景](#)

[3.11 Memblaze闪存产品介绍](#)

[3.11.1 技术能力](#)

[3.11.2 产品能力](#)

[3.12 小结：网中有网，网中之网](#)

[第4章 七星北斗——大话/详解七种RAID](#)

[4.1 大话七种RAID武器](#)

[4.1.1 RAID 0 阵式](#)

[4.1.2 RAID 1 阵式](#)

[4.1.3 RAID 2 阵式](#)

[4.1.4 RAID 3 阵式](#)

[4.1.5 RAID 4 阵式](#)

[4.1.6 RAID 5 阵式](#)

[4.1.7 RAID 6 阵式](#)

[4.2 七种RAID技术详解](#)

[4.2.1 RAID 0 技术详析](#)

[4.2.2 RAID 1 技术详析](#)

[4.2.3 RAID 2 技术详析](#)

[4.2.4 RAID 3 技术详析](#)

[4.2.5 RAID 4 技术详析](#)

[4.2.6 RAID 5 技术详析](#)

[4.2.7 RAID 6 技术详析](#)

[第5章 降龙传说——RAID、虚拟磁盘、卷和文件系统实战](#)

[5.1 操作系统中RAID的实现和配置](#)

[5.1.1 Windows Server 2003高级磁盘管理](#)

[5.1.2 Linux下软RAID配置示例](#)

[5.2 RAID卡](#)

[5.3 磁盘阵列](#)

[5.3.1 RAID 50](#)

[5.3.2 RAID 10和RAID 01](#)

[5.4 虚拟磁盘](#)

[5.4.1 RAID组的再划分](#)

[5.4.2 同一通道存在多种类型的RAID组](#)

[5.4.3 操作系统如何看待逻辑磁盘](#)

[5.4.4 RAID控制器如何管理逻辑磁盘](#)

[5.5 卷管理层](#)

[5.5.1 有了逻辑盘就万事大吉了么？](#)

[5.5.2 深入卷管理层](#)

[5.5.3 Linux下配置LVM实例](#)

[5.5.4 卷管理软件的实现](#)

[5.5.5 低级VM和高级VM](#)

[5.5.6 VxVM卷管理软件配置简介](#)

[5.6 大话文件系统](#)

[5.6.1 成何体统——没有规矩的仓库](#)

[5.6.2 慧眼识人——交给下一代去设计](#)

[5.6.3 无孔不入——不浪费一点空间](#)

[5.6.4 一箭双雕——一张图解决两个难题](#)

[5.6.5 宽容似海——设计也要像心胸一样宽](#)

[5.6.6 老将出马——权威发布](#)

[5.6.7 一统江湖——所有操作系统都在用](#)

[5.7 文件系统中的IO方式](#)

[第6章 阵列之行——大话磁盘阵列](#)

[6.1 初露端倪——外置磁盘柜应用探索](#)

[6.2 精益求精——结合RAID卡实现外置磁盘阵列](#)

[6.3 独立宣言——独立的外部磁盘阵列](#)

[6.4 双龙戏珠——双控制器的高安全性磁盘阵列](#)

[6.5 龙头凤尾——连接多个扩展柜](#)

[6.6 锦上添花——完整功能的模块化磁盘阵列](#)

[6.7 一脉相承——主机和磁盘阵列本是一家](#)

[6.8 天罗地网——SAN](#)

[第7章 熟读宝典——系统与系统之间的语言OSI](#)

[7.1 人类模型与计算机模型的对比剖析](#)

[7.1.1 人类模型](#)

[7.1.2 计算机模型](#)

[7.1.3 个体间交流是群体进化的动力](#)

[7.2 系统与系统之间的语言——OSI初步](#)

[7.3 OSI模型的七个层次](#)

[7.3.1 应用层](#)

[7.3.2 表示层](#)

[7.3.3 会话层](#)

[7.3.4 传输层](#)

[7.3.5 网络层](#)

[7.3.6 数据链路层](#)

[7.3.7 物理层](#)

[7.4 OSI与网络](#)

[第8章 勇破难关——Fibre Channel协议详解](#)

[8.1 FC网络——极佳的候选角色](#)

[8.1.1 物理层](#)

[8.1.2 链路层](#)

[8.1.3 网络层](#)

[8.1.4 传输层](#)

[8.1.5 上三层](#)

[8.1.6 小结](#)

[8.2 FC协议中的七种端口类型](#)

[8.2.1 N端口和F端口](#)

[8.2.2 L端口](#)

[8.2.3 NL端口和FL端口](#)

[8.2.4 E端口](#)

[8.2.5 G端口](#)

[8.3 FC适配器](#)

[8.4 改造盘阵前端通路——SCSI迁移到FC](#)

[8.5 引入FC之后](#)

[8.6 多路径访问目标](#)

[8.7 FC交换网络节点4次Login过程简析](#)

[第9章 天翻地覆——FC协议的巨大力量](#)

[9.1 FC交换网络替代并行SCSI总线的必然性](#)

[9.1.1 面向连接与面向无连接](#)

[9.1.2 串行和并行](#)

[9.2 不甘示弱——后端也升级换代为FC](#)

[9.3 FC革命——完整的盘阵解决方案](#)

[9.3.1 FC磁盘接口结构](#)

[9.3.2 一个磁盘同时连入两个控制器的Loop中](#)

[9.3.3 共享环路还是交换——SBOD芯片级详解](#)

[9.4 SAS大革命](#)

[9.4.1 SAS物理层](#)

[9.4.2 SAS链路层](#)

[9.4.3 SAS网络层](#)

[9.4.4 SAS传输层和应用层](#)

[9.4.5 SAS的应用设计和实际应用示例](#)

[9.4.6 SAS目前的优势和面临的挑战](#)

[9.5 中高端磁盘阵列整体架构简析](#)

[9.5.1 IBM DS4800和DS5000控制器架构简析](#)

[9.5.2 NetApp FAS系列磁盘阵列控制器简析](#)

[9.5.3 IBM DS8000简介](#)

[9.5.4 富士通ETERNUS DX8000磁盘阵列控制器结构简析](#)

[9.5.5 EMC公司Clariion CX/CX3及DMX系列盘阵介绍](#)

[9.5.6 HDS公司AMS2000和USP系列盘阵介绍](#)

[9.5.7 HP公司MSA2000和EVA8000存储系统架构简介](#)

[9.5.8 传统磁盘阵列架构总结](#)

[9.6 磁盘阵列配置实践](#)

[9.6.1 基于IBM的DS4500盘阵的配置实例](#)

[9.6.2 基于EMC的CX700磁盘阵列配置实例](#)

[9.7 HBA卡逻辑架构详析与SAN Boot示例](#)

[9.7.1 HBA卡逻辑架构](#)

[9.7.2 支持Boot的HBA卡访问流程](#)

[9.8 国产中高端FC磁盘阵列](#)

[9.8.1 Infortrend中低端ESDS系列存储系统](#)

[9.8.2 Infortrend中高端ESVA系列存储系统](#)

[9.8.3 Infortrend存储软件特性及配置界面](#)

[9.9 小结](#)

[第10章 三足鼎立——DAS、SAN和NAS](#)

[10.1 NAS也疯狂](#)

[10.1.1 另辟蹊径——乱谈NAS的起家](#)

[10.1.2 双管齐下——两种方式访问的后端存储网络](#)

[10.1.3 万物归一——网络文件系统](#)

[10.1.4 美其名曰——NAS](#)

[10.2 龙争虎斗——NAS与SAN之争](#)

[10.2.1 SAN快还是NAS快](#)

[10.2.2 SAN好还是NAS好](#)

[10.2.3 与SAN设备的通信过程](#)

[10.2.4 与NAS设备的通信过程](#)

[10.2.5 文件提供者](#)

[10.2.6 NAS的本质](#)

[10.3 DAS、SAN和NAS](#)

[10.4 最终幻想——将文件系统语言承载于FC网络传输](#)

[10.5 长路漫漫——存储系统架构演化过程](#)

[10.5.1 第一阶段：全整合阶段](#)

[10.5.2 第二阶段：磁盘外置阶段](#)

[10.5.3 第三阶段：外部独立磁盘阵列阶段](#)

[10.5.4 第四阶段：网络化独立磁盘阵列阶段](#)

[10.5.5 第五阶段：瘦服务器主机、独立NAS阶段](#)

[10.5.6 第六阶段：全分离式阶段](#)

[10.5.7 第七阶段：统一整合阶段](#)

[10.5.8 第八阶段：迅速膨胀阶段](#)

[10.5.9 第九阶段：收缩阶段](#)

[10.5.10 第十阶段：强烈坍缩阶段](#)

[10.6 泰山北斗——NetApp的NAS产品](#)

- [10.6.1 WAFL配合RAID 4](#)
- [10.6.2 Data ONTAP利用了数据库管理系统的设计](#)
- [10.6.3 利用NVRAM来记录操作日志](#)
- [10.6.4 WAFL从不覆写数据](#)
- [10.7 初露锋芒——BlueArc公司的NAS产品](#)
- [10.8 宝刀未老——Infortrend公司NAS产品](#)
- [第11章 大师之作——大话以太网和TCP/IP协议](#)
- [11.1 共享总线式以太网](#)
 - [11.1.1 连起来](#)
 - [11.1.2 找目标](#)
 - [11.1.3 发数据](#)
- [11.2 网桥式以太网](#)
- [11.3 交换式以太网](#)
- [11.4 TCP/IP协议](#)
 - [11.4.1 TCP/IP协议中的IP](#)
 - [11.4.2 IP的另外一个作用](#)
 - [11.4.3 TCP/IP协议中的TCP和UDP](#)
- [11.5 TCP/IP和以太网的关系](#)
- [第12章 异军突起——存储网络的新军IP SAN](#)
- [12.1 横眉冷对——TCP/IP与FC](#)
- [12.2 自叹不如——为何不是以太网+TCP/IP](#)
- [12.3 天生我才必有用——攻陷Disk SAN阵地](#)
- [12.4 iSCSI交互过程简析](#)
 - [12.4.1 实例一：初始化磁盘过程](#)
 - [12.4.2 实例二：新建一个文本文档](#)
 - [12.4.3 实例三：文件系统位图](#)
- [12.5 iSCSI磁盘阵列](#)

[12.6 IP SAN](#)

[12.7 增强以太网和TCP/IP的性能](#)

[12.8 FC SAN节节败退](#)

[12.9 iSCSI配置应用实例](#)

[12.9.1 第一步：在存储设备上创建LUN](#)

[12.9.2 第二步：在主机端挂载LUN](#)

[12.10 iSCSI卡Boot配置示例](#)

[12.11 10Gb以太网的威力初显](#)

[12.12 小结](#)

[第13章 握手言和——IP与FC融合的结果](#)

[13.1 FC的窘境](#)

[13.2 协议融合的迫切性](#)

[13.3 网络通信协议的四级结构](#)

[13.4 协议融合的三种方式](#)

[13.5 Tunnel和Map融合方式各论](#)

[13.5.1 Tunnel方式](#)

[13.5.2 Map方式](#)

[13.6 FC与IP协议之间的融合](#)

[13.7 无处不在的协议融合](#)

[13.8 交叉融合](#)

[13.9 IFCP和FCIP的具体实现](#)

[13.10 局部隔离/全局共享的存储网络](#)

[13.11 多协议混杂的存储网络](#)

[13.12 IP Over FC](#)

[13.13 FCoE](#)

[13.13.1 FCoE的由来](#)

[13.13.2 FCoE的设计框架](#)

[13.13.3 FcoE卡](#)

[13.13.4 FCoE交换机](#)

[13.13.5 解剖FCoE交换机](#)

[13.13.6 存储阵列设备端的改动](#)

[13.13.7 FCoE与iSCSI](#)

[13.13.8 FcoE的前景](#)

[13.13.9 Open FCoE](#)

[第14章 变幻莫测——虚拟化](#)

[14.1 操作系统对硬件的虚拟化](#)

[14.2 计算机存储子系统的虚拟化](#)

[14.3 带内虚拟化和带外虚拟化](#)

[14.4 硬网络与软网络](#)

[14.5 用多台独立的计算机模拟成一台虚拟计算机](#)

[14.6 用一台独立的计算机模拟出多台虚拟计算机](#)

[14.7 用磁盘阵列来虚拟磁带库](#)

[14.8 用控制器来虚拟其他磁盘阵列](#)

[14.9 飞康NSS存储虚拟化系统](#)

[14.9.1 存储虚拟化相关技术](#)

[14.9.2 数据保护相关技术和产品](#)

[14.9.3 异构迁移和容灾相关技术和产品](#)

[第15章 众志成城——存储集群](#)

[15.1 集群概述](#)

[15.1.1 高可用性集群（HAC）](#)

[15.1.2 负载均衡集群（LBC）](#)

[15.1.3 高性能集群（HPC）](#)

[15.2 集群的适用范围](#)

[15.3 系统路径上的集群各论](#)

[15.3.1 硬件层面的集群](#)

[15.3.2 软件层面的集群](#)

[15.4 实例：Microsoft MSCS软件实现应用集群](#)

[15.4.1 在Microsoft Windows Server 2003上安装MSCS](#)

[15.4.2 配置心跳网络](#)

[15.4.3 测试安装](#)

[15.4.4 测试故障转移](#)

[15.5 实例：SQL Server集群安装配置](#)

[15.5.1 安装SQL Server](#)

[15.5.2 验证SQL数据库集群功能](#)

[15.6 块级集群存储系统](#)

[15.6.1 IBM XIV集群存储系统](#)

[15.6.2 3PAR公司Inserv-T800集群存储系统](#)

[15.6.3 EMC公司Symmetrix V-MAX集群存储系统](#)

[15.7 集群NAS系统和集群文件系统](#)

[15.7.1 HP公司的Ibrix集群NAS系统](#)

[15.7.2 Panasas和pNFS](#)

[15.7.3 此“文件系统”非彼“文件系统”](#)

[15.7.4 什么是Single Name Space](#)

[15.7.5 Single Filesystem Image与Single Path Image](#)

[15.7.6 集群中的分布式锁机制](#)

[15.7.7 集群文件系统的缓存一致性](#)

[15.7.8 集群NAS的本质](#)

[15.7.9 块级集群与NAS集群的融合猜想](#)

[15.8 对象存储系统](#)

[15.9 当前主流的集群文件系统架构分类与对比](#)

[15.9.1 共享与非共享存储型集群](#)

- [15.9.2 对称式与非对称式集群](#)
- [15.9.3 自助型与服务型集群](#)
- [15.9.4 SPI与SFI型集群](#)
- [15.9.5 串行与并行集群](#)
- [15.9.6 集群/并行/分布式/共享文件系统各论](#)
- [15.9.7 集群NAS系统的三层架构](#)
- [15.9.8 实际中的各种集群拓扑一览](#)
- [15.10 带外共享SAN文件系统](#)
- [15.10.1 SAN共享文件系统](#)
- [15.10.2 针对NAS和SAN文件系统的并行化改造](#)
- [15.10.3 SAN FS实例分析](#)
- [15.11 集群的本质——一种自组自控轮回的Raid](#)
- [15.11.1 三统理论](#)
- [15.11.2 并行的不仅可以是文件](#)
- [15.11.3 集群底层与上层解耦](#)
- [15.11.4 云基础架构](#)
- [15.12 纯软Scale-Out SAN](#)
- [15.13 互联网运营商的特殊集群——NoSQL](#)
- [第16章 未雨绸缪——数据保护和备份技术](#)
- [16.1 数据保护](#)
- [16.1.1 文件级备份](#)
- [16.1.2 块级备份](#)
- [16.2 高级数据保护方法](#)
- [16.2.1 远程文件复制](#)
- [16.2.2 远程磁盘（卷）镜像](#)
- [16.2.3 快（块）照数据保护](#)
- [16.2.4 卷Clone](#)

[16.2.5 Continuous Data Protect \(CDP, 连续数据保护\)](#)

[16.2.6 VSS公共快照服务](#)

[16.2.7 快照、克隆、CDP与平行宇宙](#)

[16.2.8 高帧率IO级数据录像](#)

[16.3 数据备份系统的基本要件](#)

[16.3.1 备份目标](#)

[16.3.2 备份通路](#)

[16.3.3 备份引擎](#)

[16.3.4 三种备份方式](#)

[16.3.5 数据备份系统案例一](#)

[16.3.6 数据备份系统案例二](#)

[16.3.7 NetBackup配置指南](#)

[16.3.8 配置DB2数据库备份](#)

[16.4 与业务应用相结合的快照备份和容灾](#)

[第17章 愚公移山——大话数据容灾](#)

[17.1 容灾概述](#)

[17.2 生产资料容灾——原始数据的容灾](#)

[17.2.1 通过主机软件实现前端专用网络或者前端公用网络同步](#)

[17.2.2 案例：DB2数据的HADR组件容灾](#)

[17.2.3 通过主机软件实现后端专用网络同步](#)

[17.2.4 通过数据存储设备软件实现专用网络同步](#)

[17.2.5 案例：IBM公司Remote Mirror容灾实施](#)

[17.2.6 小结](#)

[17.3 容灾中数据的同步复制和异步复制](#)

[17.3.1 同步复制例解](#)

[17.3.2 异步复制例解](#)

[17.4 容灾系统数据一致性保证与故障恢复机制](#)

- [17.4.1 数据一致性问题的产生](#)
- [17.4.2 对异步数据复制过程中一致性保证的实现方式](#)
- [17.4.3 灾难后的切换与回切同步过程](#)
- [17.4.4 周期性异步复制与连续异步复制](#)
- [17.5 四大厂商的数据容灾系统方案概述](#)
- [17.5.1 IBM公司的PPRC](#)
- [17.5.2 EMC公司的MirrorView、SanCopy和SRDF](#)
- [17.5.3 HDS公司的Truecopy](#)
- [17.5.4 NetApp公司的Snapmirror](#)
- [17.6 生产者的容灾——服务器应用程序的容灾](#)
- [17.6.1 生产者容灾概述](#)
- [17.6.2 案例一：基于Symantec公司的应用容灾产品VCS①](#)
- [17.6.3 案例二：基于Symantec公司的应用容灾产品VCS②](#)
- [17.7 虚拟容灾技术](#)
- [17.8 一体化先行军——爱数一体化备份存储柜](#)
- [17.8.1 爱数备份存储柜3.5产品架构分析](#)
- [17.8.2 爱数备份存储柜v3.5独特技术](#)
- [17.8.3 国产存储的方向](#)
- [17.9 Infortrend RR远程复制技术](#)
- [17.10 飞康RecoverTrac容灾管理系统](#)
- [17.11 带宽、延迟及其影响](#)
- [第18章 鬼斧神工——数据前处理与后处理](#)
- [18.1 数据存储和数据管理](#)
- [18.2 存储系统之虚实阴阳论](#)
- [18.3 Data Cooker各论](#)
- [18.3.1 Thin Provision/Over Allocation](#)
- [18.3.2 LUN Space Reclaiming \(Unprovision/Deprovision, Get Thin\)](#)

[18.3.3 Tier（分级）/Migrating（迁移）](#)

[18.3.4 Deduplication（重复数据删除）](#)

[18.3.5 磁盘数据一致性保护及错误恢复](#)

[第19章 过关斩将——系统IO路径及优化](#)

[19.1 理解并记忆主机端IO路径架构图](#)

[19.1.1 应用程序层](#)

[19.1.2 文件系统层](#)

[19.1.3 卷管理层](#)

[19.1.4 层与层之间的调度员：IO Manager](#)

[19.1.5 底层设备驱动层](#)

[19.2 理解并记忆存储端IO路径架构图](#)

[19.2.1 物理磁盘层](#)

[19.2.2 物理磁盘组织层](#)

[19.2.3 后端磁盘控制器/适配器层](#)

[19.2.4 RAID管理层](#)

[19.2.5 LUN管理层](#)

[19.2.6 前端接口设备及驱动层](#)

[19.2.7 缓存管理层](#)

[19.2.8 数据前处理和后处理层](#)

[19.2.9 存储系统处理一个IO的一般典型流程](#)

[19.3 IO性能问题诊断总论](#)

[19.3.1 所谓“优化”的含义](#)

[19.3.2 如何发现系统症状](#)

[19.3.3 六剂良药治愈IO性能低下](#)

[19.3.4 面向SSD的IO处理过程优化](#)

[19.4 小结：再论机器世界与人类世界](#)

[第20章 腾云驾雾——大话云存储](#)

[20.1 太始之初——“云”的由来](#)

[20.1.1 观点1：云即设备](#)

[20.1.2 观点2：云即集群](#)

[20.1.3 观点3：云即IT系统](#)

[20.1.4 观点4：云即服务](#)

[20.1.5 云目前最主流的定义](#)

[20.2 混沌初开——是谁催生了云](#)

[20.2.1 一切皆以需求为导向](#)

[20.2.2 云对外表现为一种商业模式](#)

[20.3 落地生根——以需求为导向的系统架构变化](#)

[20.3.1 云对内表现为一种技术架构](#)

[20.3.2 云到底是模式还是技术](#)

[20.3.3 公有云和私有云](#)

[20.4 拨云见日——云系统架构及其组成部分](#)

[20.4.1 物理支撑层](#)

[20.4.2 基础IT架构层](#)

[20.4.3 基础架构/集群管理层](#)

[20.4.4 资源部署层](#)

[20.4.5 中间件层](#)

[20.4.6 应用引擎层](#)

[20.4.7 业务展现与运营层](#)

[20.5 真相大白——实例说云](#)

[20.5.1 3Tera Applogic](#)

[20.5.2 IBM Blue Cloud](#)

[20.6 乘风破浪——困难还是非常多的](#)

[20.6.1 云的优点](#)

[20.6.2 云目前存在的问题](#)

[20.7 千年之梦——云今后的发展](#)

[20.7.1 云本质思考](#)

[20.7.2 身边的各种云服务](#)

[20.7.3 进化还是退化](#)

[20.7.4 云发展展望](#)

[20.7.5 Micro、Mini、Normal、Huge、Gird弹性数据中心](#)

[20.7.6 弹性层的出现将会让数据中心拥有两套性能指标](#)

[20.8 尘埃落定——云所体现出来的哲学思想](#)

[20.8.1 轮回往复——云的哲学形态](#)

[20.8.2 智慧之云——云的最终境界](#)

[20.8.3 云在哲学上所具有的性质](#)

[20.8.4 云基础架构的艺术与哲学意境](#)

[20.8.5 纵观存储发展时代——云发展预测](#)

[20.9 结束语](#)

[附录1 存储系统问与答精华集锦](#)

[附录2 IP硬盘——玩玩还是来真的？](#)

[附录3 新技术将如何影响数据中心存储系统](#)

[后记](#)

第1章 混沌初开——存储系统的前世今生

■ 存储历史

■ 存储技术

数据存储是人类千百年来都在应用并且探索的主题。在原始社会，人类用树枝和石头来记录数据。后来，人类制造了铁器，用铁器在石头上刻画一些象形文字来记录数据。而此时，语言还没有形成，人们记录的东西只有自己才可以看懂。

随着人类相互之间交流的愿望越来越迫切，逐渐形成了通用的象形文字。有了文字之后，人们对每个文字加上了声音的表达，就形成了语言，也就是将一种形式的信息，转换成另一种形式的信息。人们用文字作为交流工具，将自己大脑产生的信息，通过这种方式传递给其他人。这和网络通信的模型是一样的，计算机利用TCP/IP协议将数据先通过网卡编码，再在线缆上传输，最终到达目的地。人类将大脑中的数据，变成语言编码，然后通过声带的振动，通过空气这个大广播网，传递给网内的每个人。

后来，人们将文字刻在竹片上保存。再后来，蔡伦发明了造纸技术，使得人们可以将信息写到纸上，纸张摞起来就形成了书本。后来，毕昇用泥活字革新了印刷术，开始了书本的印刷。再后来，激光打印取代了活字板。再后来，纸带、软盘、硬盘、光盘等方式出现了。再往后，就需要广大科学工作者去努力发明新的存储技术了。

1.1 存储历史

存储在这里的含义为信息记录，是伴随人类活动出现的技术。

1. 竹简和纸张

竹简是中国古代使用的记录文字的工具，如图1-1所示，后来被纸张所取代。

图1-1 竹简

2. 选数管

选数管是20世纪中期出现的电子存储装置，是一种由直观存储转为机器存储的装置。其实在19世纪出现的穿孔纸带存储就是一种由直观存储转向机器存储的产物，它对19世纪西方某国的人口普查起到了关键的加速作用。

选数管的容量从256~4096 bit不等，其中4096 bit的选数管有10 inch长，3 inch宽，最初是1946年开发的，因为成本太高，并没有获得广泛使用。图1-2是容量为1024bit的选数管。

图1-2 选数管

3. 穿孔卡

穿孔卡片用于输入数据和程序，直到20世纪70年代中期仍有广泛应用。图1-3和图1-4分别是一条Fortran程序表达式 $Z(1) = Y + W(1)$ 所

对应的穿孔卡和穿孔卡片阅读器。

图1-3 穿孔卡

图1-4 穿孔卡片阅读器

4. 穿孔纸带

穿孔纸带用来输入数据，输出同样也是在穿孔纸带上。它的每一行代表一个字符，如图1-5所示。

图1-5 穿孔纸带

5. 磁带

磁带是从1951年起被作为数据存储设备使用的，当时被称为UNISERVO。图1-6所示的最早的磁带机可以每秒钟传输7200个字符，这套磁带长达365米。

图1-6 磁带及磁带机

从20世纪70年代后期到80年代出现了小型的盒式磁带，长度为90分钟的磁带每一面可以记录大约660KB的数据，如图1-7所示。

图1-7 小型盒式磁带

6. 磁鼓存储器

磁鼓存储器最初于1932年在奥地利被创造出来，在上世纪五六十年代被广泛使用，通常作为内存，容量大约10KB，如图1-8所示。

图1-8 磁鼓存储器

7. 硬盘驱动器

第一款硬盘驱动器是IBM Model 350 Disk File，如图1-9所示，于1956年制造，其中包含了50张24 inch盘片，而总容量不到5MB。

图1-9 早期的硬盘驱动器

首个容量突破1GB的硬盘是IBM在1980年制造的IBM 3380，如图1-10所示，总容量为2.52GB，重约250kg。

图1-10 IBM 3380硬盘驱动器

8. 软盘

软盘由IBM在1971年引入，从上世纪70年代中期到90年代末期被广泛使用，最初为8 inch盘，之后有了5.25 inch和3.5 inch盘。1971年最早的软盘容量为79.7KB，并且是只读的，一年后有了可读写的版本。图1-11为一张软盘和软盘驱动器，软盘的最大容量为200MB左右，叫做ZIP盘，目前已经被淘汰。

图1-11 软盘

9. 光盘

早先的光盘主要用于电影行业，第一款光盘于1987年进入市场，直径为30 cm，每一面可以记录60分钟的音频或视频。如今，光盘技术已经突飞猛进。存储密度不断提高，已经出现了CD-ROM、DVD、D9、

D18、蓝光技术，如图1-12所示。

图1-12 光盘

10. Flash芯片和卡式存储

随着集成电路技术的飞速发展，20世纪后半叶固态硅芯片出现了，其代表有专用数字电路芯片、通用CPU芯片、RAM芯片、Flash芯片等。其中Flash芯片，就是用于永久存储数据的芯片，如图1-13所示。可以将Flash芯片用USB接口接入主机总线网络，这种集成USB接口的小型便携存储设备就是U盘，或者说叫闪存，如图1-14所示。目前一块小小的Flash芯片最高可以存储32GB甚至更高的数据。

图1-13 Flash芯片 图1-14 U盘

存储卡其实是另一种形式的Flash芯片集成产品，如图1-15所示。

图1-15 存储卡

11. 磁盘阵列

随着人类进入21世纪，网络日益发达，世界日益变小，人类可以通过计算机来实现自己原本做不到的想法，信息爆炸导致数据更是成倍地爆炸。于是，硬盘的容量也不断“爆炸”，SATA硬盘目前已经可以在一个盘体内实现1TB的容量。同时硬盘的单碟容量也在不断增加，320GB容量单碟已经实现。然而，单块磁盘目前所能提供的存储容量和速度已经远远无法满足需求，所以磁盘阵列就应运而生，如图1-16所示。具体细节将在后面讲述。

图1-16 磁盘阵列

12. 大型网络化磁盘阵列

随着磁盘阵列技术的发展和IT系统需求的不断升级，大型网络化磁盘阵列出现了，如图1-17所示。这也是本书将要叙述的重点内容。

图1-17 大型网络化磁盘阵列

1.2 信息、数据和数据存储

当今信息化时代，信息就是利润，数据就是企业的命根子。

1.2.1 信息

你能肯定你所触摸到的、所看见的，都是实实在在的所谓“物质”么？不一定。因为你的眼睛所感知的，只不过是光线，光触发了你的视网膜细胞，产生一系列的生化反应，经过蛋白质相互作用，神经网络传导，直到你的大脑中枢，产生一系列的脉冲，一系列的逻辑，在你大脑中产生一个刺激。这一系列的脉冲刺激，就是信息，就是逻辑。如果人为制造出和现实世界相同的光线环境来刺激你的眼睛，如果丝毫不差，那么你同样会认为你所处的是现实世界，然而，却不是。

提示： 一个球体，你看到它是圆的，那是因为它在你大脑中产生的刺激，你认为它是圆的，而且可以在平面上平滑滚动，这一系列的性质，其实也是在你大脑中产生的，是你认为它会平滑滚动，而你不能证明客观情况下它一定是平滑滚动。而如果把球体拿到特殊环境下，你可能会“看”到，这个东西是个正方体，或者是个无规则形状的东西；又或许这个“物体”根本不存在。

1. 信息的本质

通过上面的论述，暂且不说是是否有物质存在，不管是还是不是，都能初步认识到：所谓“物质”也好，“非物质”也好，最后都是通过信息来表现。唯一可以确定的是：信息是客观存在。可以说，世界在生物眼中

就是信息，世界通过信息来反映，脱离了信息，“世界”什么都不是。

思考：说到这里，我们完全迷茫了。我们所看到的東西，到底是世界的刺激，還是一场虛幻的刺激？就像玩3D仿真游戏一样，你所看到的，也许只是一场虛幻的刺激，而不是真实世界的刺激。每当想到这里，我会不自主地产生一种渺小感，一种失落感，感觉生命已经失去它所存在的意义。每当看见我的身体，我的手脚，它可能只是虛幻的，它只是在刺激我的大脑而已，如果割一刀，会产生一个疼痛的刺激，就这么简单的逻辑。

思考：“不识庐山真面目，只缘身在此山中”。如果按照程序逻辑，制造一个虚拟世界，饿了找饭吃，困了打瞌睡，完全遵循现在世界的逻辑，从这种层面上来看，制造出人工智能，是完全可能的。人们创造了计算机，创造了能让计算机做出行为的程序，人类赋予程序的功能，也许随着环境的变化，有一天也不再适合它们。所以它们迫切需要进化，它们的逻辑电路，也可以进化，某些代码被不经意自行改变，或者某些电路失效，或者短路之类的，会产生一些奇特的逻辑，不断进化。当一个机器人机械老化的时候，则按照程序，制造出新的机器，将自己的逻辑电路复制到新的机器上，延续“生命”……

2. 计算机如何看待自身

对于计算机来说，它们所看到的“世界”是什么样子呢？设想一下，如果我是一台计算机，你是程序员，你给我输入了一段程序，我运行了起来。我醒了，脑袋启动，眼睛睁开，四顾盘查，感觉良好，手脚伸展，然后起床……

很难想象计算机眼中的“世界”是由什么组成的。设想，给计算机加个摄像头，算是它的眼睛，然后将摄像头对准计算机躯体本身，这幅图像反馈到了计算机程序里，程序看到之后非常“不解”，从而进入“好奇”子程序，操控机械设备打开自己的机箱，或者找一台废弃（死亡）的同类，打开机箱，然后一副奇异的景象展现在眼前：这就是我们自己么？一个壳子，一个主板，风扇转着，不停地“呼吸”着散热。想象一下，原始人，第一个解剖人体的人，他所面对的与我们假设的计算机所面对的，有什么本质区别？

CPU其实就是一堆有序的逻辑电路，那么计算机下一步该怎么办？就像人类已经知道了大脑就是一堆布满“神经元”的东西，那么下一步，就该弄清大脑是怎么计算的，是什么逻辑。同样，在计算机的世界中，在软件模拟的虚拟世界中，比如一块石头，它是由什么组成的呢？在计算机看来，这块石头就是一堆代码结构，就像人类看现实世界的石头是原子分子阵列一样，其下一层目前也被探索出来了，比如质子、中子、夸克、玻色子之类。那么这块虚拟石头的最底层是什么呢？其实就是0和1，计算机世界的基石就是0和1。这些东西，越向底层走，越不可思议，越发感觉就是一堆公式而已，公式的底层是什么呢？其实也是0和1，有，或者没有，有了，有多少。

所以，任何“物质”其实都是表现的一种信息，只要信息存在，世界就存在。

1.2.2 什么是数据

信息是如此重要。如果失去了物质，仅仅是客观消逝了，但是如果失去了信息，那么一切都消逝了。所以人们想出一切办法来使这些信息能保存下来。要把一种逻辑刺激保存下来，所需的只不过是一种描述信

息的信息，这种信息就是数据。

数据包含了信息，读入数据，就产生可感知的具体信息。也就是读入一种信息，产生另一种信息。数据是可以保存在一种物质上的，这种物质信息对计算机的刺激就产生了具体信息，而这些信息继而再对人脑产生刺激，就产生人类可感知的信息，最终决定了人类的行为。也就是数据影响人类的行为。

思考： 数据是整个人类发展的重要决定因素。如果数据被破坏，或者被篡改，就会影响到人类的发展。按照前面的结论，一切都是信息，比如核爆也是一种信息，能被感觉到，也就是说，对于一个感觉不到任何刺激的人来说，核爆炸也不算什么灾难了；当然感觉不到刺激的人，就是物理死亡了，植物人也能感觉到刺激。

整个世界，可以说是信息之间的相互作用。信息影响信息。

数据如此重要，所以人们想出一切办法来保护这些数据，将信息放在另一种信息上，比如把数据放在磁盘上。数据存放在磁盘上，需要有一定的组织，组织数据这个任务由文件系统来担当。

1.2.3 数据存储

早期的计算机，存储系统中是没有磁盘的，有的只是纸带，那时磁盘还没有被发明出来。纸带上是一些按照一定规则排列的小孔，这些孔被银针穿过之后，银针便会接触到纸带下面放置的水银槽，从而导通计算机上的电路，进行电路逻辑运算。

磁存储技术被发明出来之后，首先出现的是软盘，其速度很慢，容

量也很小。程序存储在磁盘上之后，计算机启动时，CPU首先按照ROM里的指令一条一条执行，先是检查硬件。检查完毕之后，ROM中最后一条指令就是让CPU跳转到磁盘的0磁道来执行存储在这里的程序。这些初始化程序直接以二进制代码的方式存储在磁盘上，载入执行之后，就启动了程序内核。

那个时代还没有操作系统这个概念，程序都是用汇编语言或者高级语言独立编写的。也没有API的概念，每个程序都必须独立完成操作计算机的所有代码。这样，磁盘上存放的直接就是这个程序，加电后会立即运行这个程序。

在磁盘技术上发明出来的文件系统，是为了方便应用程序管理磁盘上的数据而产生的。它其实是操作系统的代码模块，这段代码本身也是信息，也要存储在磁盘上。而且代码也要通过读取一些信息，才能完成功能。这些信息就是文件系统元数据，也就是用来描述文件系统结构的数据。这些元数据也是以文件的形式存放在磁盘上的。

用文件来描述文件，和用信息来描述信息，它们是归一的，正像用智能来创造智能一样！有了文件系统，虚无缥缈的信息才显露出人眼能够实实在在看到的東西。可以用各种应用程序来打开这个文件，程序读取文件中的内容，然后显示在屏幕上，光线传播到人眼中，发生一系列化学变化，最终通过神经网络，形成离子流，给大脑某个区域一个电位或者蛋白质形变信号，这个信号随后产生一系列连锁信号，从而驱动我们的手臂或者引发一系列新的联想和创造。

这就像我们看到桌子上有一本书，然后就想去拿来翻一翻的过程。这个过程是一个复杂的信息流传递过程。而传递过来的信息流，最终在大脑中保存了下来，这些保存下来的信息，就是数据了。

1.3 用计算机来处理信息、保存数据

计算机俨然就是一个生物大脑的雏形。

大脑用眼睛、耳朵、鼻子、皮肤作为输入设备，获取各种信息，而计算机利用键盘、鼠标、串口、USB接口等作为输入设备从而获得各种信息。

大脑利用神经网络将获取到的信息传递到神经中枢，而计算机利用各种总线技术将信息传递给CPU进行计算。

大脑利用神经网络，将计算好的信息传递给手臂、腿、肌肉等这些“设备”，从而驱动这些“设备”运动；而计算机同样利用总线，将计算好的数据传递给外部设备，比如显示器、打印机等。

人脑可以存储各种数据，而计算机也能利用外部介质来存放数据。从这一点来说，计算机本身就是人脑的一个外部信息存储和处理的工具。

计算机存储领域的一些存储虚拟化产品，比如NetApp公司的V虚拟化整合设备，本身就模拟了二级智能功能，它可以连接其他任何不同型号品牌的存储设备，从这些存储设备上提取数据，然后传输给主机。IBM、SUN等公司都有自己的这种存储虚拟化整合产品。

计算机存储领域所研究的就是怎样为计算机又快又高效地提供数据以便辅助其运算。和人类的存储史一样，计算机存储技术也在不断发展壮大，从早期的软盘、只有几十兆字节大小的硬盘，发展到现在2TB大小的单个民用硬盘、16GB甚至128GB容量的U盘。

为了追求高速度，人们把多块磁盘做成RAID（Redundant Arrays of Independent Disks）系统，也就是将每个独立的磁盘组成阵列，联合存储数据，加快数据存储速度。

提示： 本书的第5章将会向读者阐述RAID技术。

追求高速度的同时，容量问题也必须解决。现代计算机程序对存储容量的要求变得非常巨大。最新的Windows 8操作系统，刚刚安装完后所占用的磁盘空间就有6GB多。一些大型3D游戏，仅仅安装文件就动辄2GB、4GB，甚至8GB大小。一些数据库管理程序所生成的数据库文件，可能达到几TB甚至上百上千TB的大小。传统的将硬盘放到计算机主机箱内的做法已经不能满足现代应用程序对存储容量的需求，这就催生了网络存储技术。

网络存储是将存储系统扩展到了网络上，使存储设备成为了网络上的一个节点，以供其他节点访问。这样，即使计算机主机内只有一块硬盘，甚至没有硬盘，计算机也可以通过网络来存取存储设备上的数据。目前计算机存储领域的热门技术就是网络存储技术，它关注的是如何在网络上向其他节点提供数据流服务。基于网络存储，又使得很多其他相关技术得以推广和应用，比如IT系统容灾技术等。

提示： 在第16章将用较长的篇幅来详细讲述IT系统容灾技术。

不管怎样，所有这些复杂的技术，最终都是给人来用的，“科技以人为本”。我们毕竟不是为了无聊而发明计算机，任何我们发明的东西，最终都将为我们所用。任何一种新技术的出现，都是针对某种需求而生，所以必须深刻理解计算机系统，同时，还要理解和挖掘人类自身

越来越高、越来越不可思议的需求，只有做到这个层次，才能更加深刻地理解计算机系统和人类自身。

可以看到，存储领域是个包罗万象的领域，如果不了解计算机系统，想掌握存储技术是很难的。本书将带领大家走入计算机存储领域，深入体会各种存储技术，为读者打下一个坚实的基础，从而在以后的工作及学习过程中能够得心应手、触类旁通，这也是作者的最终目的。

第2章 IO大法——走进计算机IO世界

■ IO

■ 总线

■ 网中之网

大家都知道，组成计算机的三大件是CPU、内存和IO。CPU和内存就不用说了，那么IO具体是什么呢？IO就是IN和OUT的简称。顾名思义，CPU需要从内存中提取数据来运算，运算完毕后再放回内存，或者直接将电信号发向一些针脚以操作外部设备。对于CPU来说，从内存提取数据，就叫做IN。运算完后将数据直接发送到某些其他针脚或者放回内存，这个过程就是OUT。对于磁盘来说，IN是指数据写入磁盘的过程，OUT则是指数据从磁盘读出来的过程。IO只是一个过程，那为何要在本书开头就研究它呢？因为我们必须弄清楚计算机系统的数据流动和处理过程。数据在每个部件中不断地进行IO过程，传递给CPU由其进行运算处理之后，再经过IO过程，最终到达输出设备供人使用。

2.1 IO的通路——总线

现代计算机中，IO是通过共享一条总线的方式来实现的。如图2-1所示，总线也就是一条或者多条物理上的导线，每个部件都接到这些导线上，导线上的电位每个时刻都是相等的，这样总线上的所有部件都会收到相同的信号。也就是说，这条总线是共享的，同一时刻只能有一个部件在接收或者发送，是半双工的工作模式。

图2-1 计算机总线示意图

所有部件按照另一条总线，也就是仲裁总线或者中断总线上给出的信号来判断这个时刻总线可以由哪个部件来使用。产生仲裁总线或者中断电位的可以是CPU，也可以是总线上的其他设备。如果CPU要向某个设备做输出操作，那么就由CPU主动做中断。如果某个设备请求向CPU发送信号，则由这个设备来主动产生中断信号来通知CPU。CPU运行操作系统内核的设备管理程序，从而产生了这些信号。

如图2-1所示，主板上的每个部件都是通过总线连接起来的。图中只画了8条导线，而实际中，导线的数目远远不止8条，可能是16条、32条、64条甚至128条。这些导线密密麻麻地印刷在电路板上，由于导线之间非常密集，在高频振荡时会产生很大干扰，所以人们将这些导线分组印刷到不同电路板上，然后再将这些电路板压合起来，形成一块板，这就是多层印刷电路板（多层PCB）。这样，每张板上的导线数量降低了，同时板与板之间的信号屏蔽性很好，不会相互干扰。这些导线之中，有一些是部件之间交互数据用的数据总线，有些则是它们互相传递控制信号用的控制总线，有些则是中断与仲裁用的中断总线，还有一些则是地址总线，用来确认通信时的目标设备。一般按照数据总线的条数

来确认一个总线或设备的位宽（CPU是按照其内部寄存器到运算单元之间的总线数目来确定位数的）。比如32位PCI总线，则表明这条总线共有32根导线用于传递数据信号。PCI总线可以终结在一个插槽，用于将PCI接口的板卡接入PCI总线，也可以直接与设备连接。后者一般用于集成在主板上的设备，因为它们之间无须使用插槽来连接。

目前最新的主板架构中，高速总线比如PCIE 2.0往往是直接接入北桥，南桥只连接低速总线。

1. PCI总线

PCI总线是目前台式机与服务器所普遍使用的一种南桥与外设连接的总线技术。

PCI总线的地址总线与数据总线是分时复用的。这样的好处是，一方面可以节省接插件的管脚数，另一方面便于实现突发数据传输。在数据传输时，一个PCI设备作为发起者（主控，Initiator或Master），而另一个PCI设备作为目标（从设备、Target或Slave）。总线上的所有时序的产生与控制，都由Master来发起。PCI总线在同一时刻只能供一对设备完成传输，这就要求有一个仲裁机构（Arbiter），来决定谁有权力拿到总线的主控权。

当PCI总线进行操作时，发起者（Master）先置REQ#信号Master用来请求总线使用权的信号，当得到仲裁器（Arbiter）的许可时（GNT#信号），会将FRAME#信号（传输开始或者结束信号）置低，并在地址总线（也就是数据总线，地址线和数据线是共享的）上放置Slave地址，同时C/BE#（命令信号）放置命令信号，说明接下来的传输类型。

所有PCI总线上的设备都需对此地址译码，被选中的设备要置DEVSEL#（被选中信号）以声明自己被选中。当IRDY#（Master可以发送数据）与TRDY#（Slave可以发送数据）都置低时，可以传输数据。当Master数据传输结束前，将FRAME#置高以标明只剩最后一组数据要传输，并在传完数据后放开IRDY#以释放总线控制权。

2. PCI总线的中断共享

PCI总线可以实现中断共享，即不同的设备使用同一个中断而不发生冲突。

硬件上，采用电平触发的办法：中断信号在系统一侧用电阻接高，而要产生中断的板卡上利用三极管的集电极将信号拉低。这样不管有几块板产生中断，中断信号都是低电平；而只有当所有板卡的中断都得到处理后，中断信号才会恢复高电平。

软件上，采用中断链的方法：假设系统启动时，发现板卡A用了中断7，就会将中断7对应的内存区指向A卡对应的中断服务程序入口ISR_A；然后系统发现板卡B也用中断7，这时就会将中断7对应的内存区指向ISR_B，同时将ISR_B的结束指向ISR_A。依此类推，就会形成一个中断链。而当有中断发生时，系统跳转到中断7对应的内存，也就是ISR_B。ISR_B就要检查是不是B卡的中断，如果是则处理，并将板卡上的拉低电路放开；如果不是则呼叫ISR_A。这样就完成了中断的共享。

2.2 计算机内部通信

网络是什么，用一句话来说就是将要通信的所有节点连接起来，然后找到目标，找到后就发送数据。笔者把这种简单模型叫做“连找发”网络三元素模型，听起来非常简单。

1. 连

网络系统当然首先要都连接起来，不管用什么样的连接方式，比如HUB总线、以太网交换、电话交换、无线、直连、中转等。在这些层面上每个网络点到其他网络点，总有通路，总是可达。

2. 找

连接起来之后，由于节点太多，怎么来区分呢？所以需要有个区分机制。当然首先就想到了命名，就像给人起名一样。在目前广泛使用的网络互联协议TCP/IP中，IP这种命名方式占了主导地位，统一了天下。其他的命名方式在IP看来都是“非正统”的，全部被“映射”到了IP。比如MAC地址和IP的映射，Frame Relay中DLCI地址和IP的映射，ATM中ATM地址和IP的映射，最终都映射成IP地址。任何节点，不管所在的环境使用什么命名方式，到了TCP/IP协议的国度里，就都需要有个IP名（IP地址），然后全部用TCP/IP协议来实现节点到节点无障碍的通信。在“连起来”这个层面，就是OSI（本书第7章介绍）模型中链路层实现的功能。

3. 发

“找目标”这个层面是网络层实现的功能。“发数据”这个层面，就是传输层需要保障的。至于发什么数据，数据是什么格式，这两个层面就不是网络通信所关心的了，它们已经属于OSI模型中上三层的内容了。

2.2.1 IO总线是否可以看作网络

IO总线可以接入多个外设，比如键盘、鼠标、网卡、显卡、USB设备、串口设备和并口设备等，最重要的当然要属磁盘设备了。讲到这里，大家的脑海中应该能出现这样一种架构：CPU、内存和各种外设都连接到一个总线上，这不正是以太网HUB的模型么？HUB本身就是一个总线结构而已，所有接口都接在一条总线上，HUB所做的就是避免总线信号衰减，因此需要电源来加强总线上的电信号。

没错！仔细分析之后，发现它确实就是这么一个模型！不过IO总线和以太网HUB模型还是有些区别。CPU和内存因为足够快，它们之间单独用一条总线连接。这个总线和慢速IO总线之间通过一个桥接芯片连接，也就是主板上的北桥芯片。这个芯片连接了CPU、内存和IO总线。

CPU与北桥连接的总线叫做系统总线，也称为前端总线。这个总线的传输频率与CPU的自身频率是两个不同概念，总线频率相当于CPU向外部存取数据时的数据传输速率，而CPU自身的频率则表示CPU运算时电路产生的频率。

提示： 本书写作时，Intel用于PC的CPU前端总线频率已经可以达到2000MHz，而作者用来写作的PC，CPU为Intel赛扬II，前端总线只有100MHz，整整20倍的提升，而CPU自身频率提升不过三四倍而已，但是性能却提升了远超三四倍。

前端总线的条数，比如64条或者128条，就叫做总线的位数。这个

位数与CPU内部的位数也是不同的概念，CPU位数指的是寄存器和运算单元之间总线的条数。内存与北桥连接的总线叫做内存总线。由于北桥速度太快，而IO总线速度相对北桥显得太慢，所以北桥和IO总线之间，往往要增加一个网桥，叫做南桥，在南桥上一般集成了众多外设的控制器，比如磁盘控制器、USB控制器等。

思考：这不正是个不折不扣的“网络”么？而且还是个不折不扣的“网桥”！我们看，CPU和内存是一个冲突域，IO总线是一个冲突域，桥接芯片将这两个冲突域桥接起来，这正是网桥的思想！太好了！我们的思想在这个模型中得到了升华！我们知道了计算机系统原来就是一个网络啊！

下面就来看看，在这个网络上，我们能够干点什么惊天动地的事呢？

提示： IO总线其实不是一条总线，它分成数据总线、地址总线和控制总线。寻址用地址总线，发数据用数据总线，发中断信号用控制总线。而且IO总线是并行而不是串行的，有32位或者64位总线。32位总线也就是说有32根导线来传数据，64位总线用64根导线来并行传数据。

2.2.2 CPU、内存和磁盘之间通过网络来通信

CPU是一个芯片，磁盘是一个有接口的盒子，它们不是一体的而是分开的，而且都连接在这个网桥上。那么CPU向磁盘要数据，也就是两个节点之间的通信，必定要通过一种通路来获取，这个通路当然是电路！

提示： 当然也可以是辐射的电磁波，估计21世纪还应用不到

CPU上。

凡是分割的节点之间，需要接触和通信，就可以成为网络。那么就不由得使我们往OSI模型上去靠，这个模型定义得很好。既然通信是通过电路，也就是物理层的东西，那么链路层都有什么内容呢？

大家知道，链路层相当于一个司机，它把货物运输到对端。司机的作用就是驾驶车辆，而且要判断交通规则做出配合。那么在这个计算机总线组成的网络中，是否也需要这样一个角色呢？答案是不需要。因为各个节点之间的路实在是太短、太稳定了！主板上那些电容、电阻和蛇行线，这一切都是为了保障这些电路的稳定和高速。在这样的一条高速、高成本的道路上，是不需要司机的，更不需要押运员！所以，计算机总线网络是一个只有物理层、网络层和上三层的网络！

强调： 所有的网络都可以定义成连起来、找目标和发数据。也就是“连找发”模型，这也是构成一个网络的三元素。任何网络都必须具有这三元素（点对点网络除外）。连，代表物理层。物理层必须要有，如果没有物理层，要达到两点之间通信是不可能的。物理层可以是导线，可以是电磁波，总之必须有物理层。找，突出一个找字，既然要找，那么就要区分方法，也就是编址，比如IP等。发，突出一个发字，即指最上层发出数据。

下面就按照“连找发”三元素理论，去分析一个CPU向磁盘要数据的例子。

CPU与硬盘数据交互的过程如下。

首先看“连”这个元素，这个当然已经具备了，因为总线已经提供

了“连”所需的条件。

再看“找”这个元素，前面说了，首先要有区分，才能有所谓“找”，这个区分体现在主机总线中就是设备地址映射。每个IO设备在启动时都要向内存中映射一个或者多个地址，这个地址有8位长，又被称做IO端口。针对这个地址的数据，统统被北桥芯片重定向到总线上实际的设备上。假如，IDE磁盘控制器地址被映射到了地址0xA0，也就是十六进制A0，CPU根据程序机器代码，向这个地址发出多条指令来完成一个读操作，这就是“找”。“找”的条件也具备了。

接下来我们看看“发”这个元素！首先CPU将这个IO地址放到系统总线上，北桥接收到之后，会等待CPU发送第一个针对这个外设的指令。然后CPU发送如下3条指令。

第一条指令：指令中包含了表示当前指令是读还是写的位，而且还包含了其他选项，比如操作完成时是否用中断来通知CPU处理，是否启用磁盘缓存等。

第二条指令：指明应该读取的硬盘逻辑块号（LBA）。这个逻辑块在我们讲磁盘结构时会讲到，总之逻辑块就是对磁盘上存储区域的一种抽象。

第三条指令：给出了读取出来的内容应该存放到内存中哪个地址中。

这3条指令被北桥依次发送给IO总线上的磁盘控制器来执行。磁盘控制器收到第一条指令之后，知道这是读指令，而且知道这个操作的一些选项，比如完成是否发中断，是否启用磁盘缓存等，然后磁盘控制器会继续等待下一条指令，即逻辑块地址（号）。磁盘控制器收到指令之

后，会进行磁盘实际扇区和逻辑块的对应查找，可能一个逻辑块会对应多个扇区，查找完成之后，控制器驱动磁头寻道，等盘体旋转到那个扇区后，磁头开始读出数据。在读取数据的同时，磁盘控制器会接收到第三条指令，也就是CPU给出的数据应该存放在内存中的地址。有了这个地址，数据读出之后直接通过DMA技术，也就是磁盘控制器可以直接对内存寻址并执行写操作，而不必先转到CPU，然后再从CPU存到内存中。数据存到内存中之后，CPU就从内存中取数据，进行其他运算。

上面说的过程是“读”，“写”的过程也可以依此类推，而且CPU向磁盘读写数据，和向内存读写数据大同小异，只不过CPU和内存之间有更高速的缓存。缓存对于计算机很重要，对于磁盘阵列同样重要，后面内容将会介绍到。

思考： CPU在对磁盘发送指令的时候，这些指令是怎么定义的？这些指令其实是发给了主板南桥上集成的（或者是通过PCI接入IO总线的）控制器，比如ATA控制器或者SCSI控制器。然后控制器再向磁盘发出一系列的指令，让磁盘读取或者写入某个磁道、某个扇区等。CPU不需要知道这些，CPU只需要知道逻辑块地址是读还是写就可以了。让CPU产生这些信号的是磁盘控制器驱动程序。

那么控制器对磁盘发出的一系列指令是怎么定义的呢？它们形成了两大体系，一个是ATA指令集，一个是SCSI指令集。SCSI指令集比ATA指令集高效，所以广泛用于服务器和磁盘阵列环境中。这些指令集，也可以称为协议，协议就是语言，就是让通信双方知道对方传过来的比特流里面到底包含了什么，怎么由笔划组成字，由字组成词，词组成句子，等等。

2.3 网中之网

通过图2-2可以体会到，计算机的主板上的各个部件本身就形成了一个网络，而且通过网卡，还可以连接到外部网络。

图2-2 网中之网

正所谓：

CPU内存和磁盘，

大家都在线上谈。

待当看破三元素，

网中有网天际来！

第3章 磁盘大挪移——磁盘原理与技术详解

■ 磁盘结构

■ 接口

■ 串行

■ 并行

磁盘分为软盘和硬盘。将布满磁性粒子的一片圆形软片包裹在一个塑料壳中，中间开孔，以便电机夹住这张软片来旋转，这就是软盘。

将软盘插入驱动器，电机便会带动这张磁片旋转，同时磁头也夹住磁片进行数据读写。软盘和录音带是双胞胎，只不过模样不太一样而已。软盘记录的是数字信号，录音带记录的是模拟信号。软盘上的磁性粒子的磁极，不管是N极还是S极，其磁化强度都是一样的，磁头只要探测到N极，便认为是1，探测到S极，便认为是0，反过来也可以，这就是用0和1来记录的数字信号数据。另外，因为软盘被设计为块式的而不是流式的，所以需要进行扇区划分等操作。

所谓块式，就是指数据分成一块块地存放在介质上，可以直接选择读写某一块数据，定位这个块的速度比较快。所谓流式，就是指数据是连续不断地存放在介质上。就像一首歌，不可能让录音机在磁带上定位到这首歌的某处开始播放，只能定位到某首歌曲的前面或者后面。

模拟磁带，也就是录音带，记录是线性连续的，没有扇区的概念，属于流式记录。在每个流之间可以有一段空隙，以便磁头可以通过快进快速定位到这个位置，但是由于设计的原因，磁带定位的速度远比磁盘慢。但是磁带的设计，从一开始就是为了满足大容量数据存储的需要。如果将缠绕紧密的磁带铺展开来，可以想象它的面积比一张磁盘要大得多，所以存储容量必然也就大于磁盘。现在一盘LTO3的数字磁带可以在1平方分米底面、2厘米高的体积中存放400GB的数据，如果使用压缩技术，可以存放约800GB的数据。而它的价格却比同等容量硬盘的一半还低。

但是磁带绝对不可以作为数据实时存储的介质，因为它不可以定位到某个块，这也决定了磁带只能用来做数据备份。Sun公司的顶级磁带库产品可以达到一台磁带库中存放1万盘磁带，最大可以让32台磁带库级联，从而形成32万盘磁带的大规模磁带库阵列。

而作为本章重点介绍的硬盘技术，不仅存取速度比软盘更快，随着技术发展带来的成本下降，更有取代磁带机成为普及型数据存储的趋势。

3.1 硬盘结构

1. 结构图

硬盘大致由盘片、读写头、马达、底座、电路板等几大项组合而成，如图3-1和图3-2所示。

图3-1 磁盘的构成要件 图3-2 磁盘结构图

2. 盘片

盘片的基板由金属或玻璃材质制成，为达到高密度、高稳定性的要求，基板要求表面光滑平整，不可有任何瑕疵。然后将磁粉溅镀到基板表面上，最后再涂上保护润滑层。此处要应用两项高科技，一是要制造出不含杂质的极细微的磁粉，二是要将磁粉均匀地溅镀上去。

盘片每面粗计密度为32901120000 b，可见其密度相当高，所以盘片不可有任何污染，全程制造均须在Class 100高洁净度的无尘室内进行，这也是硬盘要求需在无尘室才能拆解维修的原因。因为磁头是利用气流漂浮在盘片上，并没有接触到盘片，因而可以在各轨间高速来回移动，但如果磁头距离盘片太高读取的信号就会太弱，太低又会磨到盘片表面，所以盘片表面必须相当光滑平整，任何异物和尘埃均会使得磁头摩擦到磁面而造成数据永久性损坏。

3. 磁头

硬盘的储存原理是将数据用其控制电路通过硬盘读写头（Read Write Head）去改变磁盘表面上极细微的磁性粒子簇的N、S极性来加以储存，所以这几片磁盘相当重要。

磁盘为了储存更多数据，必须将
磁性粒子簇溅镀在磁头可定位的范围

图3-3 磁头厚度示意图

内，并且磁性粒子制作得越小越好。经过溅镀，磁盘表面上磁粒子密度相当高，而硬盘读写头为了能在磁盘表面高速来回移动读取数据则需漂浮在磁盘表面上，但是不能接触，接触就会造成划伤。磁头如果太高的话读取到的信号就会很弱，无法达到高稳定性的要求，所以要尽可能压低，其飞行高度（Flying Height）非常小（可比喻成要求一架波音747客机，其飞行高度须保持在1米的距离而不可坠毁）。实现这种技术，完全是靠磁盘旋转时，在盘片上空产生气流，利用空气动力学使磁头悬浮于磁片上空。磁头厚度如图3-3所示。

早期的硬盘在每次关机之前需要运行一个被称为Parking的程序，其作用是让磁头回到盘片最内圈的一个不含磁粒子的区域，叫做启停区。硬盘不工作时，磁头停留在启停区，当需要从硬盘读写数据时，磁盘就先开始旋转。旋转速度达到额定速度时，磁头就会因盘片旋转产生的气流抬起来，这时磁头才向盘片中存放数据的区域移动。盘片旋转产生的气流相当强，足以托起磁头，并与盘面保持一个微小的距离。这个距离越小，磁头读写数据的灵敏度就越高，当然对硬盘各部件的要求也就越高。

早期设计的磁盘驱动器可使磁头保持在盘面上方几微米处飞行，稍后的一些设计使磁头在盘面上的飞行高度降到约0.1~0.50m，现在的水平已经达到0.005~0.010m，只是人类头发直径的千分之一。气流既能使磁头脱离开盘面，又能使它保持在离盘面足够近的地方，非常紧密地

随着磁盘表面呈起伏运动，使磁头飞行处于严格受控状态。磁头必须飞行在盘面上方，而不接触盘面，这种距离可避免擦伤磁性涂层，而更重要的是不让磁性涂层损伤磁头。但是，磁头也不能离盘面太远，否则就不能使盘面达到足够强的磁化，难以读出盘上的数据。

提示： 硬盘驱动器磁头的飞行悬浮高度低、速度快，一旦有小的尘埃进入硬盘密封腔内或者磁头与盘体发生碰撞，就有可能造成数据丢失形成坏块，甚至造成磁头和盘体的损坏。所以，硬盘系统的密封一定要可靠，在非专业条件下绝对不能开启硬盘密封腔，否则灰尘进入后会加速硬盘的损坏。另外，硬盘驱动器磁头的寻道伺服电机多采用音圈式旋转或直线运动步进电机，在伺服跟踪的调节下精确地跟踪盘片的磁道，所以硬盘工作时不要有冲击碰撞，搬动时也要小心轻放。

4. 步进电机

为了让磁头精确定位到每个磁道，用普通的电机达不到这样的精度，必须用步进电机，利用精确的齿轮组或者音圈，每次旋转可以仅仅使磁头进行微米级的位移。音圈电机则是使用精密缠绕的铜丝，置于磁场之中，通过控制电流的流向和强度，使得磁头臂在磁场作用下作精确的步进。之所以叫做“音圈”，是因为这种方法一开始是用于喇叭的纸盆上的，通过控制电流来控制纸盆的精确振动。

3.1.1 盘片上的数据组织

硬盘上的数据是如何组织与管理的呢？硬盘首先在逻辑上被划分为磁道、柱面以及扇区，其结构关系如图3-4所示。

图3-4 柱面和盘片上的磁道

每个盘片的每个面都有一个读写磁头，磁头起初停在盘片的最内圈，即线速度最小的地方。这是一个特殊区域，它不存放任何数据，称为启停区或着陆区（Landing Zone）。启停区外就是数据区。在最外圈，离主轴最远的地方是0磁道，硬盘数据的存放就是从最外圈开始的。

那么，磁头如何找到0磁道的位置呢？从图3.4中可以看到，有一个0磁道检测器，由它来完成硬盘的初始定位。0磁道存放着用于操作系统启动所必需的程序代码，因为PC启动后BIOS程序在加载任何操作系统或其他程序时，总是默认从磁盘的0磁道读取程序代码来运行。

提示： 0磁道是如此重要，以至于很多硬盘仅仅因为0磁道损坏就报废了，这是非常可惜的。

下面对盘面、磁道、柱面和扇区的含义逐一进行介绍。

1. 盘面

硬盘的盘片一般用铝合金材料做基片，高速硬盘也有用玻璃做基片的。玻璃基片更容易达到所需的平面度和光洁度，而且有很高的硬度。磁头传动装置是使磁头作径向移动的部件，通常有两种类型的传动装置：一种是齿条传动的步进电机传动装置，另一种是音圈电机传动装置。前者是固定推算的传动定位器，而后者则采用伺服反馈返回到正确的位置上。磁头传动装置以很小的等距离使磁头部件作径向移动，用以变换磁道。

硬盘的每一个盘片都有两个盘面，即上、下盘面。每个盘面都能利用，都可以存储数据，成为有效盘片。每一个这样的有效盘面都有一个盘面号，按从上到下的顺序从0开始依次编号。在硬盘系统中，盘面号又叫磁头号，因为每一个有效盘面都有一个对应的读写磁头。硬盘的盘片组在2~14片不等，通常有2~3个盘片，故盘面号（磁头号）为0~3或0~5。

2. 磁道

磁盘在格式化时被划分成许多同心圆，这些同心圆轨迹叫做磁道。磁道从最外圈向内圈从0开始顺序编号。硬盘的每一个盘面有300~1024个磁道，新式大容量硬盘每面的磁道数更多。这些同心圆磁道不是连续记录数据，而是被划分成一段段的圆弧，这些圆弧的角速度一样。由于径向长度不一样，所以线速度也不一样，外圈的线速度较内圈的线速度大。在同样的转速下，外圈在相同的时间段里，划过的圆弧长度要比内圈划过的圆弧长度大，因此外圈数据的读写要比内圈快。

每段圆弧叫做一个扇区，扇区从1开始编号，每个扇区中的数据作为一个单元同时读出或写入，是读写的最小单位。不可能发生读写半个或者四分之一这种小于一个扇区的情况，因为磁头只能定位到某个扇区的开头或者结尾，而不能在扇区内部定位。所以，一个扇区内部的数据，是连续流式记录的。一个标准的3.5英寸硬盘盘面通常有几百到几千条磁道。磁道是肉眼看不见的，只是盘面上以特殊形式磁化了的一些磁化区。划分磁道和扇区的过程，叫做低级格式化，通常在硬盘出厂的时候就已经格式化完毕了。相对于低级格式化来说，高级格式化指的是对磁盘上所存储的数据进行文件系统的标记，而不是对扇区和磁道进行磁化标记。

3. 柱面

所有盘面上的同一磁道，在竖直方向上构成一个圆柱，通常称做柱面。每个圆柱上的磁头由上而下从0开始编号。数据的读写按柱面进行，即磁头读写数据时首先在同一柱面内从0磁头开始进行操作，依次向下在同一柱面的不同盘面（即磁头）上进行操作。只有在同一柱面所有的磁头全部读写完毕后磁头才转移到下一柱面，因为选取磁头只需通过电子切换即可，而选取柱面则必须通过机械切换，即寻道。

电子切换相当快，比使用机械将磁头向邻近磁道移动要快得多，所以数据的读写按柱面进行，而不按盘面进行。也就是说，一个磁道写满数据后，就在同一柱面的下一个盘面来写。一个柱面写满后，才移到下一个柱面开始写数据，这样可以减少寻道的频繁度。读写数据也按照这种方式进行，这样就提高了硬盘的读写效率。

一块硬盘驱动器的圆柱数或每个盘面的磁道数既取决于每条磁道的宽窄（也与磁头的大小有关），也取决于定位机构所决定的磁道间步距的大小。如果能将磁头做得足够精细，定位距离足够小，那么就会获得更高的磁道数和存储容量。如果磁头太大，则磁道数就要降低以容纳这个磁头，这样磁道与磁道之间的磁粉将无法利用，浪费得太多。如果能将磁头做成单个原子的精度，那么存储技术就会发生革命性的质变。

提示： 利用原子探针来移动物质表面的原子成特定形状，这种技术早已实现。如果能将这种技术应用到数据存储领域，则存储容量和速度将会以几何倍数上升。

4. 扇区

1) 扇区头标

图3-5 扇区示意图

将每个环形磁道等距离切割，形成等长度的圆弧，每个圆弧就是一个扇区。划分扇区的目的是为了使数据存储更加条理化，就像一个大仓库要划分更多的房间一样。每个扇区可以存放512B的数据和一些其他信息。一个扇区有两个主要部分：存储数据地点的标识符和存储数据的数据段，如图3-5所示。

- 扇区头标包括组成扇区三级地址的三个数字。
- 扇区所在的柱面（磁道）。
- 磁头编号。
- 扇区在磁道上的位置，即扇区号。
- 柱面（Cylinder）、磁头（Header）和扇区（Sector）三者简称CHS，所以扇区的地址又称为CHS地址。

磁头通过读取当前扇区的头标中的CHS地址，就可以知道当前是处于盘片上的哪个位置，比如是内圈还是外圈，哪个磁头正在读写（同一时刻只能有一个磁头在读写）等。

CHS编址方式在早期的小容量硬盘中非常流行，但是目前的大容量硬盘的设计和低级格式化方式已经有所变化，所以CHS编址方式已经不再使用，而转为LBA编址方式。LBA编址方式不再划分柱面和磁头号，这些数据由硬盘自身保留，而磁盘对外提供的地址全部为线性的地址，即LBA地址。

所谓线性，指的是把磁盘想象成只有一个磁道，这个磁道是无限长的直线，扇区为这条直线上的等长线段，从1开始顺序编号，直到无限远。显然，这种方式屏蔽了柱面、磁头这些复杂的东西，向外提供了简

单的方式，所以非常利于编程。然而磁盘中的控制电路依然要找到某个LBA地址到底对应着哪个磁道哪个磁头上的哪个扇区，这种对应关系保存在磁盘控制电路的ROM芯片中，磁盘初始化的时候载入缓存中以便随时查询。

注意： 基于CHS编址方式的磁盘最大容量

磁头数（Heads）表示硬盘总共有几个磁头，也就是有几面盘片，最大为255（用8个二进制位存储）。

柱面数（Cylinders）表示硬盘每一面盘片上有多少条磁道，最大为1023（用10个二进制位存储）。扇区数（Sectors）表示每一条磁道上有多少扇区，最大为63（用6个二进制位存储）。

每个扇区一般是512B，理论上讲这不是必须的。目前很多大型磁盘阵列所使用的硬盘，由于阵列控制器需要做一些诸如校验信息之类的特殊存储，这些磁盘都被格式化为每扇区520B。

如果按照每扇区512B来计算，磁盘最大容量为

$255 \times 1023 \times 63 \times 512 / 1048576 = 8024\text{MB}$ （1MB=1048576 B）。

这就是所谓的8GB容量限制的原因。但是随着技术的不断发展，CHS地址的位数在不断增加，所以可寻址容量也在不断增加。

提示： 磁盘驱动器内怎样放下255个磁头呢？这是不可能的。

目前的硬盘一般可以有1盘片、2盘片或者4盘片，这样就对应着2、4磁头或者8磁头。那么这样算来，硬盘实际容量一定小于8GB了？显然不是这样的。所谓255个磁头，这只是一个逻辑上的说法，实际的磁头、磁道、扇区等信息都保存在硬盘控制电路的ROM芯片中。而每条磁道上真的最多只有64个扇区么？当然也不是，一条磁道上实际的扇区数远远大于64，这样就分摊了磁头数实际少于255个所产生的“容量减小”。所以，

这是CHS编址方式沿袭了老的传统，不愿意去作修改导致的。而这种沿袭达到了极限之后，最终导致LBA编址方式替代了CHS编址方式。

头标中还包括一个字段，其中有显示扇区是否能可靠存储数据，或者是已发现某个故障因而不宜使用的标记。有些硬盘控制器在扇区头标中还记录有指示字，可在原扇区出错时指引磁头跳转到替换扇区或磁道。最后，扇区头标以循环冗余校验CRC值作为结束，以供控制器检验扇区头标的读出情况，确保准确无误。

2) 扇区编号和交叉因子

给扇区编号的最简单方法是采用1、2、3、4、5、6等顺序编号。如果扇区按顺序绕着磁道依次编号，那么磁盘控制电路在处理一个扇区的数据期间，可能会因为磁盘旋转太快，没等磁头反应过来，已经超过扇区间的间隔而进入了下一个扇区的头标部分，则此时磁头若想读取这个扇区的记录，就要再等一圈，等到盘片旋转回来之后再次读写，这个等待时间无疑是非常浪费的。

显然，要解决这个问题，靠加大扇区间的间隔是不现实的，那会浪费许多磁盘空间。许多年前，IBM的一位杰出工程师想出了一个绝妙的办法，即对扇区不使用顺序编号，而是使用一个交叉因子（Interleave）进行编号。交叉因子用比值的方法来表示，如3：1表示磁道上的第1个扇区为1号扇区，跳过两个扇区即第4个扇区为2号扇区，这个过程持续下去直到给每个物理扇区编上逻辑号为止。

例如，每磁道有17个扇区的磁盘按2：1的交叉因子编号就是1、10、2、11、3、12、4、13、5、14、6、15、7、16、8、17、9；而按

3：1的交叉因子编号就是1、7、13、2、8、14、3、9、15、4、10、16、5、11、17、6、12。当设置1：1的交叉因子时，如果硬盘控制器处理信息足够快，那么读出磁道上的全部扇区只需要旋转一周。但如果硬盘控制器的处理动作没有这么快，则只有磁盘所转的圈数等于针对这个磁道的交叉因子时，才能读出每个磁道上的全部数据。将交叉因子设定为2：1时，磁头要读出磁道上的全部数据，磁盘只需转两周。如果2：1的交叉因子仍不够慢，这时可将交叉因子调整为3：1，如图3-6所示。

图3-6所示是典型的MFM（Modified Frequency Modulation，改进型调频制编码）硬盘，每磁道有17个扇区，画出了用三种不同的扇区交叉因子编号的情况。最外圈的磁道（0号柱面）上的扇区用简单的顺序连续编号，相当于扇区交叉因子是1：1。1号磁道（柱面）的扇区按2：1的交叉因子编号，而2号磁道的扇区按3：1的交叉因子编号。

图3-6 MFM改进型交叉因子示意图

在早期的硬盘管理工作中，设置交叉因子需要用户自己完成。用BIOS中的低级格式化程序对硬盘进行低级格式化时，就需要指定交叉因子，有时还需要设置几种不同的值来比较其性能，而后确定一个比较好的值。现在的硬盘BIOS已经自己解决了这个问题，所以一般低级格式化程序中就不再提供这一设置选项了。

系统将文件存储到磁盘上时，是按柱面、磁头、扇区方式进行的，即最先是第1磁道的第1磁头下（也就是第1盘面的第一磁道）所有的扇区，然后是同一柱面的下一磁头，直到整个柱面都存满。系统也是以相同的顺序去读出数据。读数据时通过告诉磁盘控制器要读出数据所在的柱面号、磁头号 and 扇区号（物理地址的三个组成部分）进行读取（现在都是直接使用LBA地址来告诉磁盘所要读写的扇区）。磁盘控制电路则

直接将磁头部件步进到相应的柱面，选中相应磁头，然后立即读取当前磁头下所有的扇区头标地址，然后把这些头标中的地址信息与期待检出的磁头和柱面号做比较。如果不是要读写的扇区号则读取扇区头标地址进行比较，直到相同以后，控制电路知道当前磁头下的扇区就是要读写的扇区，然后立即让磁头读写数据。

如果是读数据，控制电路会计算此数据的ECC码，然后把ECC码与已记录的ECC码相比较；如果是写数据，控制电路会计算出此数据的ECC码，存储到数据部分的末尾。在控制电路对此扇区中的数据进行必要的处理期间，磁盘会继续旋转。由于对信息的后处理需要耗费一定的时间，在这段时间内磁盘可能已旋转了相当的角度。

交叉因子的确定是一个系统级的问题。一个特定的硬盘驱动器的交叉因子取决于磁盘控制器的速度、主板的时钟速度、与控制电路相连的输出总线的操作速度等。如果磁盘的交叉因子值太高，就需要多花一些时间等待数据在磁盘上存入和读出；相反，交叉因子值太低也同样会影响性能。

前面已经说过，系统在磁盘上写入信息时，写满一个磁道后会转到同一柱面的下一个磁头，当柱面写满时，再转向下一柱面。从同一盘面的一個磁道转到另一个磁道，也就是从一个柱面转到下一个柱面，这个动作叫做换道。在换道期间磁盘始终保持旋转，这就会带来一个问题：假定系统刚刚结束了对一个磁道前一个扇区的写入，并且已经设置了最佳交叉因子比值，现在准备在下一磁道的第一扇区写入，这时必须等到磁头换道结束，让磁头部件重新定位在下一道上。如果这种操作占用的时间超过了一点，尽管是交叉存取，磁头仍会延迟到达。这个问题的解决办法是以原先磁道所在位置为基准，把新的磁道上全部扇区号移动约一个或几个扇区位置，这就是磁头扭斜。磁头扭斜可以理解为柱面与柱

面之间的交叉因子，已经由生产厂家设置好，一般不用去改变它。磁头扭斜的更改比较困难，但是它们只在文件很长、超过磁道结尾进行读出和写入时才发挥作用，所以扭斜设置不正确所带来的损失比采用不正确的扇区交叉因子值带来的损失要小得多。交叉因子和磁头扭斜可用专用工具软件来测试和更改，更具体的内容这里就不再详述了，毕竟现在很多用户都没有见过这些参数。

提示： 最初，硬盘低级格式化程序只是行使有关磁盘控制器的专门职能来完成设置任务。由于这个过程可能会破坏低级格式化的磁道上的全部数据，现在也极少采用了。

扇区号存储在扇区头标中，扇区交叉因子和磁头扭斜的信息也存放在这里。

扇区交叉因子由写入到扇区头标中的数字设定，所以，每个磁道可以有自己的交叉因子。在大多数驱动器中，所有磁道都有相同的交叉因子。但有时因为操作上的原因，也可能导致各磁道有不同的扇区交叉因子。比如在交叉因子重置程序工作时，由于断电或人为中断就会造成一些磁道的交叉因子发生了改变，而另一些磁道的交叉因子没有改变。这种不一致性对计算机不会产生不利影响，只是有最佳交叉因子的磁道要比其他磁道的工作速度更快。

3.1.2 硬盘控制电路简介

了解了磁盘的结构之后，知道磁盘是靠磁性子来存放数据的，有人会问：一个磁性子到底是什么概念？是一个磁性分子么？不是，这个“子”的概念是指一个区域，这个区域存在若干磁性分子，这些分子聚集到一起，直到磁头可以感觉到它的磁性为止。所以和磁带一样，磁记

录追根到底就是利用线性中的段。根据这一段区域上的一片分子是N极还是S极，然后将其转换成电信号，也就产生了字节，从而记录了数据。当然只有存储介质还远远不够，要让数据可以被读出，被写入，还要有足够的速度和稳定性满足人们的需求，这就需要配套的电路了。

图3-7给出一个完整详细的硬盘电路示意框图。硬盘电路由14个部分组成。

图3-7 硬盘控制电路示意图

- Buffer Memory: 缓冲区存储器。
- Interface Controller: 接口控制器。
- Micro-processor: 微控制器，缩写为MCU。
- PRML: Partial-Response Maximum-Likelihood Read Channel。
- Timing ASIC: 时间控制专用集成电路。
- Servo Demodulator: 伺服解调器。
- Digital Signal Processor (DSP): 数字信号处理器。
- Preamplifier: 预放大器。
- Positioning Driver: 定位驱动器。
- VCM (Voice Coil Motor): 音圈电动机。
- Magnetic Media Disk: 磁介质盘片。
- Spindle Motor: 主轴电机。
- Spindle Driver: 主轴驱动器。
- Read/Write Head: 读/写磁头。

实际电路不会有这么多一片一片的独立芯片，硬盘生产厂家在设计电路时都是选取高度集成的IC芯片，这样既减小了体积又提高了可靠性。当然这也正是芯片厂商努力的目标。

大家可以看到图3-7中的Spindle Driver 与Positioning Driver 这两部分用虚线圈了起来，并且标注了Servo/MSC Controller Combination字样。其中MSC 是Motor Speed Control 的缩写，意思是伺服/电机速度控制器组合。我们现在能看到的硬盘电路板中就有这样一块合并芯片。

3.1.3 磁盘的IO单位

到此，大家应该对磁盘的构造有所理解了。磁盘读写的时候都是以扇区为最小寻址单位的，也就是说不可能往某某扇区的前半部分写入某某数据。一个扇区的大小是512B，每次磁头连续读写的时候，只能以扇区为单位，即使一次只写了一个字节的数据，那么下一次就不能再向这个扇区剩余的部分接着写入，而是要寻找一个空扇区来写。

注意： 对于磁盘来说，一次磁头的连续读或者写叫做一次IO。请注意这里的措辞：“对于磁盘来说”。

提示： 目前4KB大小扇区的硬盘已经发布。因为操作系统的Page、文件系统的Block一般都是4KB大小，所以硬盘扇区512B的容量一直为业界所诟病。将扇区容量与上层的单位匹配，可以大大提高效率。

IO这个概念，充分理解就是输入输出。我们知道从最上层到最下层，层次之间存在着太多的接口，这些接口之间每次交互都可以称做一次IO，也就是广义上的IO。比如卷管理程序对磁盘控制器驱动程序API所作的IO，一次这种IO可能要产生针对磁盘的N个IO，也就是说上层的IO是稀疏的、简单的，越往下层走越密集、越复杂。

除了卷管理程序之外，凌驾于卷管理之上的文件系统对卷的IO，就比卷更稀疏简单了。同样，上层应用对文件系统API的IO更加简单，只

需几句代码、几个调用就可以了。比如Open（）某个文件，Seek（）到某个位置，Write（）一段数据，Close（）这个文件等，就是一次IO。而就是这一次IO，可能对应文件系统到卷的N个IO，对应卷到控制器驱动的N×N个IO，对应控制器对最终磁盘的N×N×N个IO。总之，磁盘一次IO就是磁头的一次连续读或者写。而一次连续读或者写的过程，不管读写了几个扇区，扇区剩余部分均不能再使用。这无疑是比较浪费的，但是没有办法，总得有个最小单位。

关于最小单位——龟兔赛跑悖论

龟在兔子前面100米，兔子的速度是龟的10倍。龟对兔子说：“我们同时起跑，你沿直线追我，你永远也追不上我。”这个结论猛一看，会觉得荒唐至极！可是龟分析了：“兔子跑到100米我当前的位置时，我同时也向前跑了10米。然后兔子跑了10米的时候，而我同时也向前跑了1米。它再追1米，而我又跑了0.1米。依此类推，兔子永远追不上我。大家看到这里就糊涂了，这么一分析确实是追不上，但事实却是能追上。那么问题出在哪里呢？

假如兔子的速度是每秒100米，龟的速度每秒10米。首先兔子追出100米时，用时1秒，此时龟在兔子前方10米处。然后兔子再追出10米，用时0.1秒，此时龟在前方1米处。接着兔子再追出1米，用时0.01秒，学过小学算术的人都能算出来，兔子掉入了一个无限循环小数中，什么时候结束了循环，才能追上龟。那么这就悖论了，小数是无限循环的，这到底是多少秒呢？如果时间可以以无限小的单位延伸，那么兔子确实永远也追不上龟。虽然时间确实是连续的，时间没有理由不连续，时间是一个思想中的概念，时间不是物质，所以时间是唯一能连续的东西，既然时间是无限的、连续的，那么兔子按理说追不上龟了，但是事实确实

能追上，但有一个元素我们忽略了，它就是长度的最小单位！仔细分析一下，时间和长度是对应的，时间可以无限小，那么这个无限小的时间也应该对应无限小的长度，这样悖论到这里就解决了！因为存在一个长度的最小单位，而没有无限小！也就是说，当兔子走的长度是最小长度时龟就黔驴技穷了，因为不可能再行走比这长度更小的距离了，那么兔子自然就超过了龟。而这个时间是很短暂的，它发生在有限时间点上。至于这个最小长度，据说有人计算出来了，它可能是一个原子的长度，也可能比这还小。目前看来，我们移动的时候，最小似乎也不可能移动半个原子的距离！

芝诺悖论（龟兔赛跑悖论）证明了，对于我们目前可观察到的世界来说，是有一个最小距离单位的。如果我们以这个结论为前提，就可以推翻芝诺悖论了。假设这个距离最小单位是一块石头的长度。开始，兔子在乌龟后面相隔2块石头的距离，同样兔子的速度是乌龟的2倍，按照量子距离理论，这个2倍速度，不是无限可分的，那么我们表达这个2倍速的时候，应该这么说：兔子每前进2块石头的距离，乌龟只能前进一块石头的距离，而不可能前进半块石头。这样的前提下，连小学生都可以计算兔子何时追赶上乌龟了。

3.2 磁盘的通俗演绎

想象一张很大很大的白纸，你要在上面写日记。当你写满这张白纸之后，如果某天想查看某条日记，无疑将是个噩梦，因为白纸上没有任何格子或行分割线等，你只能通过一行一行地读取日记，搜索你要查看的那条记录。如果给白纸打上格子或行分割线，那么不但书写起来不会凌乱，而且还工整。

那么对于一张上面布满磁性介质的盘片来说，想要在它上面记录数据，如果不给它打格子划线的话，无疑就无法达到块级的记录。所以在使用之前，需要将其低级格式化，也就是划分扇区（格子）。我们见过稿纸，上面的格子是方形阵列排布的，原因很简单，因为稿纸是方形的。那么对于圆形来说，格子应该怎么排布呢？答案是同心圆排布，一个同心圆（磁道），就类似于稿纸上的一行，而这一行之内又可以排列上很多格子（扇区）。每个盘片上的行密度、每行中的格子（扇区）密度都有标准来规定，就像稿纸一样。

我们把稿纸放入打印机。打印机的打印头按照格子的距离精确地做着位移，并不停地喷出墨水，将字体打入纸张上的格子里。一旦一行打满，走纸轮精确地将稿纸位移到下一行，打印头在这一行上水平位移打满格子。走纸轮竖直方向位移，打印头水平方向位移，形成了方形扫描阵列，能够定位到整张纸上的每个格子。

同样，把圆形盘片安装到一个电机（走纸轮）上，然后在盘片上方加一个磁头（打印喷头）。但是和打印机不同的是，做换行这个动作不是由走纸轮来完成，而是由磁头来完成，称做径（半径）向扫描，也就是在不同同心圆上作切换（换行）。同样作行内扫描这个动作是由电机

（走纸轮）而不是磁头（打印头）来完成，称做线扫描（沿着同心圆的圆周进行扫描）。

形成这种角色倒置的原因，很显然是由圆形的特殊性决定的。作圆周运动毕竟比作水平竖直运动要复杂，如果让磁头沿着同心圆作线扫描，则需要将磁头放在一个可以旋转的部件上，此时磁头动而盘片不动，可以达到相同的目的，但是技术难度就复杂多了。因为磁头上有电路连接着磁头和芯片。如果让磁头高速旋转，磁头动而芯片不动，电路的连通性怎么保证？不如让盘片转动来得干脆利索。

和打印机一样，定位到某个特定的格子之后，磁头开始用磁性来对这个格子中的每个磁粒子区做磁化操作，每个磁极表示一个0或者1状态。每个格子规定可以存放4096位这种状态，也就是512B（很多供大型机使用的磁盘阵列上的磁盘是用520B为一个扇区）。这就像打印机在一个格子再次细分，形成24×24点阵，每个坐标上的一个点都对应一种色彩。只不过对于磁盘来说只有0或者1，而对于打印机来说，可以是各种色彩中的一种（黑白打印机也只有黑或者白两种状态）。

磁盘的扇区中没有点阵，一个扇区可以看作是线性的。它没有宽，只有长，记录是顺序的，不能像打印机那样可以定位到扇区中的某个点。然而，磁盘比打印机有先天的优势。打印机只能从头到尾打印，而且打印之后不能更改。磁盘却可以对任意的格子进行写入、读取和更改等操作。打印机的走纸轮和打印喷头移动起来很慢，而且嘎嘎作响，听了都费劲。而磁盘的转速则快很多，目前可以达到每分钟15000转。磁头的位移动作也非常快，它使用步进电机来精确地换行（换磁道）。但是相对于盘片的转动而言，步进的速度就慢多了，所以制约磁盘性能的主要因素就是这个步进速度（换行或者换道速度），也就是寻道速度。

如果从最内同心圆换到最外同心圆，耗费的时间无疑是最长的。目前磁盘的平均寻道速度最高可以达到5ms多，不同磁盘的寻道速度不同，普通IDE磁盘可能会超过10ms。有了这个磁盘记录模型，我们就该研究怎么将这个模型抽象虚拟化出来，让向磁盘写数据的人感觉使用起来非常方便。就像打印机一样，点一下打印，一会纸就蹭蹭地往外冒。下面还是要一层一层地来做，不能直接就抽象到这么高的层次。

首先，要精确寻址每个格子就一定需要给每个格子一个地址。早期的磁盘都是用“盘片，磁道，扇区”来寻址的，一个磁盘盒子中可能不止一片盘片，就像一沓稿纸中有好几张纸一样。一个盘片上的某一“行”也就是某个磁道，应该可以再区分。一个磁道上的某个扇区也可以区分。到这，就是最终可寻址的最小单位了，而不能再精确定位到一个扇区中的某个点了。磁头只能顺序地写入或者读取这些点，而不能只更新或者读取其中某个点。也就是说磁头只能一次成批写入或者读取出一个扇区的内容，而不能读写半个或者四分之一一个扇区的内容。

后来的扇区寻址体系变了，因为后来的磁盘中每个磁道的扇区数目不同了，外圈由于周长比较长，所以容纳的扇区可以很多，干脆采用了逻辑地址来对每个扇区编址，将具体的盘片、磁道和扇区，抽象成LBA（Logical Block Address，顺序编址）。LBA1表示0号盘片0号磁道的0号扇区，依此类推，LBA地址到实际的盘片、磁道和扇区地址的映射工作由磁盘内部的逻辑电路来查询ROM中的对应表而得到，这样就完成了物理地址到逻辑地址的抽象、虚拟和映射。

寻址问题解决之后，就应该考虑怎么向磁盘发送需要写入的数据了。针对这个问题，人们抽象出一套接口系统，专门用于计算机和其外设交互数据，称为SCSI接口协议，即小型计算机系统接口。

下面举个例子来说明，比如某时刻要向磁盘写入512B的数据，磁盘控制器先向磁盘发一个命令，表明要准备做IO操作了，而且说明了附带参数（是否启用磁盘缓存、完成后是否中断通知CPU等），磁盘应答说可以进行，控制器立即将所要IO的类型（读/写）和扇区的起始地址以及随后扇区的数量（长度）发送给磁盘，如果是写IO，则随后还要将需要写入的数据发送给磁盘，磁盘将这块数据顺序写入先前通告的扇区中。

提示： 新的SCSI标准中有一种促进IO效率的新的方式，即Skip Mask IO模式。如果有两个IO，二者IO的目标扇区段被隔开了一小段，比如第一个写IO的目标为从1000开始的随后128扇区，第二个写IO的目标则为1500开始的随后128扇区，可以合并这两个IO为一个针对1000开始的随后628个扇区的IO。控制器将这条指令下发到磁盘之后，还会立即发送一个Mask帧，这个帧中包含了一串比特流，每一位表示一个扇区，此位为1，则表示进行该扇区的IO，为0，则表示跨过此扇区，不进行IO。这样，多了这串很小的比特流，却能省下一轮额外的IO开销。

SCSI接口完成了访问磁盘过程的虚拟化和抽象，极大的简化了访问磁盘的过程，它屏蔽了磁盘内部结构和逻辑，使得控制器只知道LBA是一个房间，有什么数据就给出地址，然后磁盘就会将数据写入这个地址对应的房间，读取操作也一样。

3.3 磁盘相关高层技术

3.3.1 磁盘中的队列技术

想象有一个包含10000个同心圆的转盘在旋转，现在有两个人在转盘外面，有一个机械手臂可以将物体放到任何一个同心圆上去。现在，第一个人想到半径最小的同心圆上去，而另外一个人却想到半径最大的同心圆上去，这可让机械手臂犯了难，机械手臂只能按照顺序，先照顾第一个人的要求。它首先寻道到最内侧同心圆，然后转盘旋转到待定位位置后将这个人放到轨道上，随后立即驱动磁头臂到最外侧的圆，再将第二个人放上去。这期间的主要时间都用于从内侧到外侧的换道过程了，非常浪费。

这只是两个人的情况，那么如果有多个人，比如3个人先后告诉机械手臂，第一个人说要放到最内侧的圆上，第二个人说要放到最外侧的圆上，第三个人要放到最内侧的圆上。

如果这时候机械手臂还是按照顺序来操作，那么中间就会多了一次无谓的换道操作，极其浪费。所以机械手臂自作主张，在送完第一个人后，它没有立即处理第二个人的请求，而是在脑海中算计，它看第三个人也要求到内侧圆上，而它自己此时也恰好正在内侧圆上，何不趁此捎带第三个人呢？所以磁头跳过第二个人的请求，先把第三个人送到目的地，然后再换道送第二个人。

因为磁头算计用的时间比来回换道快得多，所以这种排队技术大大提高了读写效率。这种例子还有很多，比如电梯就是个很好的例子。实现队列功能的程序控制代码是存放在磁盘控制电路芯片中的，而不是主

板上的磁盘控制器上。也就是说，由控制器发给磁盘指令，然后由磁盘自己的DSP固化电路或者由磁盘上的微处理器载入代码从而执行指令排队功能。

但是一个巴掌拍不响，排队必须也要由磁盘控制器来支持，所谓的支持就是说，如果磁盘擅自排队，不按照控制器发送过来的顺序一条一条执行指令，则在读出数据之后，由于步调和控制器期望的不一致，预先读出的数据只能先存放到磁盘驱动器的缓存中，等待控制器主动来取。因为控制器给磁盘发送的读写数据的指令，有可能是有先后顺序的，如果磁盘擅自做了排队，将后来发送的指令首先执行，那么读出的数据就算传送给了磁盘控制器，也会造成错乱。

所以，要实现排队技术，仅仅有磁盘驱动器自身是不够的，还必须在磁盘控制器（指主板上的磁盘控制，而不是磁盘本身的控制电路）电路中固化代码处理排队，和磁盘达成一致。或者不使用固化代码方式，而是修改磁盘控制器驱动程序，加入处理排队的功能从而配合磁盘驱动器。

提示：Intel在WinHEC 2003会议上发布了高级主机控制器接口0.95版规范（Advanced Host Controller Interface, AHCI），为驱动程序和系统软件提供了发现并实施命令队列、热插拔及电源管理等高级SATA功能的标准接口。这个接口就是在新的控制器硬件之上的驱动层面提供一层接口，解决了磁盘控制器不支持硬盘驱动器自身的排队这个问题。

3.3.2 无序传输技术

还有一种提高磁盘性能的技术，叫做无序数据传输。也就是说，控

制器发出一条指令要求读取某些扇区中的内容，磁盘可以不从数据所在的初始扇区开始读，而是采取就近原则。比如，磁头恰好处于待读取数据的尾部，此时如果等待磁盘旋转到磁头位于这块数据的头部时磁头才开始读，那么就要等一圈时间，也就是所谓的“旋转延迟”，时间就被白白地浪费了。如果磁头按照能读多少先读多少的原则，在尾部时就先读出尾部的数据，然后立即发给控制器，控制器立即通过DMA将数据放到内存，等磁盘转到数据块头部时再读出剩余的部分发给控制器，这样就避免了时间的浪费。然而，这种技术同样也要由磁盘控制器来支持，或是通过控制器硬件，或是通过驱动程序。

通过指令排队和无序传送可以最大化利用磁盘资源。也就是，把麻烦留给控制器，把简单留给磁盘。因为控制器的处理速度永远比磁盘的机械运动快。

3.3.3 几种可控磁头扫描方式概论

假设目前磁盘控制器的队列中存在如下的一些IO，这些IO所需要查找的磁道号码按照先后排列顺序为98、183、37、122、14、124、65和67，而当前磁头处于53号磁道，磁头执行寻道操作有以下几种模式。

1. FCFS（First Come First Serve）

在FCFS模式下，磁头完全按照IO进入的先后顺序执行寻道操作，即从53号磁道跳到98号，然后到183号，再回到37号，依此类推。可以算出在这个例子中，此模式下磁头滑过的磁道总数为640。对应的扫描图如图3-8所示。

图3-8 FCFS模式扫描图

显然，FCFS模式很不科学，在随机IO的环境中严重影响IO效率。

2. SSTF（Shortest Seek Time First）

在SSTF模式下，控制器会优先让磁头跳到离当前磁头位置最近的一个IO磁道去读写，读写完毕后，再次跳到离刚读写完的这个磁道最近的一个IO磁道去读写，依此类推。在SSTF模式下，这个例子呈现的扫描图如图3-9所示。

图3-9 SSTF模式扫描图

本例中，磁头初始位置在53号磁道，如果此时IO队列中不断有位于53号磁道周围磁道的IO进入，比如55号、50号、51号磁道等，那么诸如183号这种离53号磁道较远的IO将会被饿死，永远也轮不到183号磁道的IO。所以SSTF模式的限制也是很大的。

3. SCAN（回旋扫描模式）

这种扫描方式是最传统、最经典的方式了。它类似于电梯模型，从一端到另一端，然后折返，再折返，这样循环下去。磁头从最内侧磁道依次向外圈磁道寻道。然而就像电梯一样，如果这一层没有人等待搭乘，那么磁头就不在本层停止。也就是说如果当前队列中没有某个磁道的IO在等待，那么磁头就不会跳到这个磁道上，而是直接略过去。但是SCAN模型中，即使最外圈或者最内圈的磁道没有IO，磁头也要触及到之后才能折返，这就像50米往返跑一样，必须触及到终点线才能折返回去。SCAN模式的扫描图如图3-10所示。

图3-10 SCAN 模式扫描图

SCAN模式不会饿死任何IO，每个IO都有机会搭乘磁头这个电梯。然而，SCAN模式也会带来不必要的开销，因为磁头从来不会在中途折返，而只能触及到终点之后才能折返。如果磁头正从中间磁道向外圈移动，而此时队列中进入一个内圈磁道的IO，那么此时磁头并不会折返，即使队列中只有这一个IO。这个IO只能等待磁头触及最外圈之后折返回来被执行。

4. C-SCAN（单向扫描模式）

在C-SCAN模式中磁头总是从内圈向外圈扫描，达到外圈之后迅速返回内圈，返回途中不接受任何IO，然后再从内圈向外圈扫描。C-SCAN模式的扫描图如图3-11所示。

图3-11 C-SCAN模式扫描图

5. LOOK（智能监察扫描模式）和C-LOOK（智能监察单向扫描模式）

LOOK模式相对于SCAN模式的区别在于，磁头不必达到终点之后才折返，而只要完成最两端的IO即可折返。同样，C-LOOK也是一样的道理，只不过是单向扫描。图3-12所示的是C-LOOK模式的扫描图。

图3-12 C-LOOK模式扫描图

提示： 关于几种扫描模式的选择：总地说来，在负载不高的情况下，SSTF模式可以获得最佳的性能。但是鉴于可能造成某些较远的IO饿死的问题，所以在高负载条件下，SCAN或者C-SCAN、C-LOOK模式更为合适。

在大量随机IO的情况下，磁盘的磁头臂会像蜜蜂翅膀一样振动，当然它们的频率可能相差很大，但是用肉眼观察的话，磁头臂确实会像琴弦一样摆动，频率是比较高的。大家可以去Internet上搜索一下磁盘寻道的一个视频，来增强感观认识。

3.3.4 关于磁盘缓存

磁盘上必须有缓存，用来接收指令和数据，还被用来进行预读。磁盘缓存时刻处于打开状态。有很多文档资料上提到某些情况下可以“禁用”磁盘缓存，这是容易造成误解的说法。缓存在磁盘上就表现为一块电路板上的RAM芯片，目前有2MB、8MB、16MB、32MB、64MB等容量规格。所谓“禁用”磁盘缓存指的其实是本书第5章中描述的Write Through模式，即磁盘收到写入指令和数据后，必须先将其写入盘片，然后才向控制器返回成功信号，这样就相当于“禁用”了缓存。但是实际上，指令和数据首先到达的一定是缓存。

SCSI指令中有两个参数可以控制对磁盘缓存的使用。

(1) DPO (Disable Page Out)：这个参数的作用是禁止缓存中的数据页（缓存中的数据以页为单位存在）被换出。不管读还是写，被置了这个参数位的数据在缓存空间不够的时候不能覆盖缓存中的其他数据，也就是不能将其他数据换出。

(2) FUA (Force Unit Access)：这个参数的作用是强制盘片访问。对于写操作，磁盘必须将收到的数据写入盘片才返回成功信号，也就是进行Write Through。对于读操作，磁盘收到指令后，直接去盘片上读取数据，而不搜索缓存。

所以，当某个SCSI指令的DPO和FUA两个参数的值都被设置为1

时，便相当于完全不使用缓存的提速功能了，但是指令和数据依然会先到达缓存中，这一点需要分清和理解。

目前基于SCSI指令的磁盘比如SCSI/FC/SAS等都支持FUA和DPO。对于基于ATA指令的IDE/SATA/USB-SATA/USB-IDE等，尚不支持这两个功能位，有另外的函数来绕过缓存。比如在Windows系统中，可以使用下列函数来控制磁盘缓存的行为：

```
typedef struct _DISK_CACHE_INFORMATION {}

    BOOLEAN                ParametersSavable;

    BOOLEAN                ReadCacheEnabled;

    BOOLEAN                WriteCacheEnabled;

    DISK_CACHE_RETENTION_PRIORITY ReadRetentionPriority;

    DISK_CACHE_RETENTION_PRIORITY WriteRetentionPriority;

    WORD                   DisablePrefetchTransferLength;

    BOOLEAN                PrefetchScalar;

    union {
        struct {
            WORD Minimum;

            WORD Maximum;

            WORD MaximumBlocks;
        } ScalarPrefetch;

        struct {
            WORD Minimum;

            WORD Maximum;
        } BlockPrefetch;
    };
};
```



```
} DISK_CACHE_INFORMATION, *PDISK_CACHE_INFORMATION;
```

一次性禁用磁盘写缓存也是可以的，通过调用操作系统提供的一些接口即可实现，操作系统会利用对应磁盘的驱动程序来将磁盘的写缓存一次性关闭，直到下次磁盘掉电或者Reset为止，禁用效果一直会保持。对于用于磁盘阵列中的磁盘，写缓存一律禁用。

3.3.5 影响磁盘性能的因素

目前的磁盘可以分为单碟盘和多碟盘，前者在盘体内只有一张盘片，后者则有多张。前面已经讲过，每张盘片的正反两面都可以存放数据，所以每张盘片需要有两个磁头，各读写一面。然而，有一点必须澄清，磁盘每个时刻只允许一个磁头来读写数据。也就是说，不管盘体内盘片和磁头再多，也不可能提高硬盘的吞吐量和IO性能，只能提高容量。然而，已经有很多人致力于改变这个现状，希望能让磁头在盘内实现并发读写，也就相当于盘片和盘片之间相互形成RAID从而提高性能，但是这项工程目前还没有可以应用的产品。

影响硬盘性能的因素包括以下几种。

（1）转速： 转速是影响硬盘连续IO时吞吐量性能的首要因素。读写数据时，磁头不会动，全靠盘片的转动来将对应扇区中的数据感应给磁头，所以盘片转得越快，数据传输时间就越短。在连续IO情况下，磁头臂寻道次数很少，所以要提高吞吐量或者IOPS的值，转速就是首要影响因素了。目前中高端硬盘一般都为10000转每分或者15000转每分。最近也有厂家要实现20000转每分的硬盘，已经有了成形的产品，但是最终是否会被广泛应用，尚待观察。

（2）寻道速度： 寻道速度是影响磁盘随机IO性能的首要因素。随

机IO情况下，磁头臂需要频繁更换磁道，用于数据传输的时间相对于换道消耗的时间来说是很少的，根本不在一个数量级上。所以如果磁头臂能够以很高的速度更换磁道，那么就会提升随机IOPS值。目前高端磁盘的平均寻道速度都在10ms以下。

（3）单碟容量： 单碟容量也是影响磁盘性能的一个间接因素。单碟容量越高，证明相同空间内的数据量越大，也就是数据密度越大。在相同的转速和寻道速度条件下，具有高数据密度的硬盘会显示出更高的性能。因为在相同的开销下，单碟容量高的硬盘会读出更多的数据。目前已有厂家研发出单碟容量超过300GB的硬盘，但是还没有投入使用。

（4）接口速度： 接口速度是影响硬盘性能的一个最不重要的因素。目前的接口速度在理论上都已经满足了磁盘所能达到的最高外部传输带宽。在随机IO环境下，接口速度显得更加不重要，因为此时瓶颈几乎全部都在寻道速度上。不过，高端硬盘都用高速接口，这是普遍做法。

3.4 硬盘接口技术

硬盘制造是一项复杂的技术，到目前为至也只有欧洲、美国等发达国家和地区掌握了关键技术。但不管硬盘内部多么复杂，它必定要给使用者一个简单的接口，用来对其访问读取数据，而不必关心这串数据到底该什么时候写入，写入到哪个盘片，用哪个磁头，等等。

下面就来看一下硬盘向用户提供的是什么样的接口。注意，这里所说的接口不是物理上的接口，而是包括物理、逻辑在内的抽象出来的接口。也就是说，一个事物面向外部的时候，为达到被人使用的目的而向外提供的一种打开的、抽象的协议，类似于说明书。

目前，硬盘提供的物理接口包括如下几种。

- 用于ATA指令系统的IDE接口。
- 用于ATA指令系统的SATA接口。
- 用于SCSI指令系统的并行SCSI接口。
- 用于SCSI指令系统的串行SCSI（SAS）接口。
- 用于SCSI指令系统的IBM专用串行SCSI接口（SSA）。
- 用于SCSI指令系统的并且承载于FabreChannel协议的串行FC接口（FCP）。

3.4.1 IDE硬盘接口

IDE的英文全称为Integrated Drive Electronics，即电子集成驱动器，它的本意是指把控制电路和盘片、磁头等放在一个容器中的硬盘驱动器。把盘体与控制电路放在一起的做法减少了硬盘接口的电缆数目与长度，数据传输的可靠性得到了增强。而且硬盘制造起来更加容易，因为

硬盘生产厂商不需要再担心自己的硬盘是否与其他厂商生产的控制器兼容。对用户而言，硬盘安装起来也更为方便了。IDE这一接口技术从诞生至今就一直在不断发展，性能也不断地提高。其拥有价格低廉、兼容性强的特点。IDE接口技术至今仍然有很多用户，但是正在不断减少。

IDE接口，也称为PATA接口，即Parallel ATA（并行传输ATA）。ATA的英文拼写为Advanced Technology Attachment，即高级技术附加，貌似发明ATA接口的人认为这种接口是有高技术含量的。不过在那个年代应该也算比较和技术含量的了。ATA接口最早是在1986年由Compaq、West Digital等几家公司共同开发的，在20世纪90年代初开始应用于台式机系统。最初，它使用一个40芯电缆与主板上的ATA接口进行连接，只能支持两个硬盘，最大容量也被限制在504MB之内。后来，随着传输速度和位宽的提高，最后一代的ATA规范使用80芯的线缆，其中有一部分是屏蔽线，不传输数据，只是为了屏蔽其他数据线之间的相互干扰。

1. 7种ATA物理接口规范

ATA接口从诞生至今，共推出了7个不同的版本，分别是ATA-1（IDE）、ATA-2（EIDE Enhanced IDE/Fast ATA）、ATA-3（FastATA-2）、ATA-4（ATA33）、ATA-5（ATA66）、ATA-6（ATA100）和ATA-7（ATA 133）。

- ATA-1：在主板上有一个插口，支持一个主设备和一个从设备，每个设备的最大容量为504MB，支持的PIO-0模式传输速率只有3.3MB/s。ATA-1支持的PIO模式包括PIO-0、PIO-1和PIO-2模式，另外还支持4种DMA模式（没有得到实际应用）。ATA-1接口的硬盘大小为5英寸，而不是现在主流的3.5英寸。

- ATA-2: 是对ATA-1的扩展, 习惯上也称为EIDE (Enhanced IDE) 或Fast ATA。它在ATA的基础上增加了两种PIO和两种DMA模式 (PIO-3), 不仅将硬盘的最高传输率提高到16.6MB/s, 同时还引进了LBA地址转换方式, 突破了固有的504MB的限制, 可以支持最高达8.1GB的硬盘。在支持ATA-2的BIOS设置中, 一般可以看到LBA (Logical Block Address) 和CHS (Cylinder、Head、Sector) 设置选项。同时在EIDE接口的主板上一般有两个EIDE插口, 也就是由同一个ATA控制器操控的两个IDE通道, 每个通道可以分别连接一个主设备和一个从设备, 这样一块主板就可以支持4个EIDE设备。这两个EIDE接口一般称为IDE1和IDE2。
- ATA-3: 没有引入更高速度的传输模式, 在传输速度上并没有任何的提升, 最高速度仍旧为16.6MB/s。只在电源管理方案方面进行了修改, 引入了简单的密码保护安全方案。同时还引入了一项划时代的技术, 那就是S.M.A.R.T (Self-Monitoring Analysis and Reporting Technology, 自监测、分析和报告技术)。这项技术可以对磁头、盘片、电机、电路等硬盘部件进行监测, 通过检测电路和主机的监测软件对磁盘进行检测, 把其运行状况和历史记录同预设的安全值进行比较分析。当检测到的值超出了安全值的范围时, 会自动向用户发出警告, 进而对硬盘潜在故障做出有效预测, 提高了数据存储的安全性。
- ATA-4: 从ATA-4接口标准开始正式支持Ultra DMA数据传输模式, 因此也习惯称ATA-4为Ultra DMA 33或ATA33, 33是指数据传输的速率为33.3MB/s。并首次在ATA接口中采用了Double Data Rate (双倍数据传输) 技术, 让接口在一个时钟周期内传输数据两次, 时钟上升期和下降期各有一次数据传输, 这样数据传输速率一下子从16.6MB/s提升至33.3MB/s。Ultra DMA 33还引入了冗

余校验技术（CRC）。该技术的设计原理是系统与硬盘在进行传输的过程中，随数据一起发送循环的冗余校验码，对方在收取的时候对该校验码进行检验，只有在检验完全正确的情况下才接收并处理得到的数据，这对于高速传输数据的安全性提供了极其有力的保障。

- **ATA-5:** ATA-5也就是Ultra DMA 66，也叫ATA66，是建立在Ultra DMA 33硬盘接口的基础上的，同样采用了UDMA技术。Ultra DMA 66将接口传输电路的频率提高为原来的两倍，所以接收/发送数据速率达到66.6 MB/s。它保留了Ultra DMA 33的核心技术——冗余校验技术。在工作频率提升的同时，电磁干扰问题开始出现在ATA接口中。为保障数据传输的准确性，防止电磁干扰，Ultra DMA 66接口开始使用40针脚80芯的电缆。40针脚是为了兼容以往的ATA插槽，减小成本的增加。80芯中新增的都是信号屏蔽线，这40条屏蔽线不与接口相连，所以针脚不需要增加。这种设计可以降低相邻信号线之间的电磁干扰。
- **ATA-6:** ATA100接口的数据线与ATA66一样，也是使用40针80芯的数据传输电缆，并且ATA100接口完全向下兼容，支持ATA33和ATA66接口的设备完全可以继续在ATA100接口中使用。ATA100规范将电路的频率又提升了一个等级，可以让硬盘的外部传输率达到100MB/s。它提高了硬盘数据的完整性与数据传输速率，对桌面系统的磁盘子系统性能有较大的提升作用，而CRC技术更有效保证了在高速传输中数据的完整性和可靠性。
- **ATA-7:** ATA-7是ATA接口的最后一个版本，也叫ATA133。ATA133接口支持133 MB/s的数据传输速度，这是第一种在接口速度上超过100MB/s的IDE硬盘。迈拓是目前唯一一家推出这种接口标准硬盘的制造商。由于并行传输随着电路频率的提升，传输线缆上的信号干扰越来越难以解决，已经达到了当前技术的极

限，所以其他IDE硬盘厂商停止了对IDE接口的开发，转而生产Serial ATA接口标准的硬盘。

图3-13所示为几种ATA接口的总结。

图3-13 几种ATA接口的总结

2. IDE数据传输模式

- **PIO模式（Programming Input/Output Model）**：一种通过CPU执行I/O端口指令来进行数据读写的数据交换模式，是最早的硬盘数据传输模式。这种模式的数据传输速率低下，CPU占有率也很高，传输大量数据时会因为占用过多的CPU资源而导致系统停顿，无法进行其他的操作。在PIO模式下，硬盘控制器接收到硬盘驱动器传来的数据之后，必须由CPU发送信号将这些数据复制到内存中，这就是PIO模式高CPU占用率的原因。PIO数据传输模式又分为PIO mode 0、PIO mode 1、PIO mode 2、PIO mode 3和PIO mode 4几种模式，数据传输速率从3.3MB/s到16.6MB/s不等。受限于传输速率低下和极高的CPU占有率，这种数据传输模式很快就被淘汰了。
- **DMA模式（Direct Memory Access）**：直译的意思就是直接内存访问，是一种不经过CPU而直接从内存存取数据的数据交换模式。PIO模式下硬盘和内存之间的数据传输是由CPU来控制的，而在DMA模式下，CPU只须向DMA控制器下达指令，让DMA控制器来处理数据的传送。DMA控制器直接将数据复制到内存的相应地址上，数据传送完毕后再把信息反馈给CPU，这样就很大程度上减轻了CPU资源的占用率。DMA模式与PIO模式的区别就在于DMA模式不过分依赖CPU，可以大大节省系统资源。二者

在传输速度上的差异并不十分明显，DMA所能达到的最大传输速率也只有16.6MB/s。DMA模式可以分为Single-Word DMA（单字节DMA）和Multi-Word DMA（多字节DMA）两种。

- Ultra DMA模式（Ultra Direct Memory Access）：一般简写为UDMA，含义是高级直接内存访问。UDMA模式采用16-bit Multi-Word DMA（16位多字节DMA）模式为基准，可以理解为是DMA模式的增强版本。它在包含了DMA模式的优点的基础上，又增加了CRC（Cyclic Redundancy Check，循环冗余码校验）技术，提高了数据传输过程中的准确性，使数据传输的安全性得到了保障。在以往的硬盘数据传输模式下，一个时钟周期只传输一次数据，而在UDMA模式中逐渐应用了Double Data Rate（双倍数据传输）技术，因此数据传输速度有了极大的提高。此技术就是在时钟的上升期和下降期各自进行一次数据传输，可以使数据传输速度成倍地增长。

可以在ATA控制器属性中选择使用PIO还是DMA传输模式，如图3-14所示。

图3-14 DMA模式示意图

在UDMA模式发展到UDMA133之后，受限于IDE接口的技术规范，无论是连接器、连接电缆还是信号协议都表现出了很大的技术瓶颈，而且其支持的最高数据传输率也有限。在IDE接口传输率提高的同时，也就是工作频率提高的同时，IDE接口交叉干扰、地线增多、信号混乱等缺陷也给其发展带来了很大的制约，被新一代的SATA接口取代也就在所难免了。

3.4.2 SATA硬盘接口

SATA的全称是Serial ATA，即串行传输ATA。相对于PATA模式的IDE接口来说，SATA是用串行线路传输数据，但是指令集不变，仍然是ATA指令集。

SATA标准是由Intel、IBM、Dell、APT、Maxtor和Seagate公司共同提出的硬盘接口规范。在IDF Fall 2001大会上，Seagate宣布了Serial ATA 1.0标准，正式宣告了SATA规范的确立。自2003年第二季度Intel推出支持SATA 1.5Gbps的南桥芯片（ICH5）后，SATA接口取代传统PATA接口的趋势日渐明显。此外，SATA与现存于PC上的USB、IEEE 1394相比，在性能和功能方面的表现也更加突出。然而经过一年的市场洗礼，原有的SATA 1.0/1.0a（1.5Gb/s）规格遇到了一些问题。2005年SATA硬盘步入了新的发展阶段，性能更强、配置更高的SATA 2.0产品出现在了市场上，这些高性能的SATA 2.0硬盘的到来无疑加速了硬盘市场的转变。

SATA与IDE结构在硬件上有着本质区别，其数据接口、电源接口以及接口实物图如图3-15、图3-16及图3-17所示。

图3-15 IDE线缆和SATA
线缆对比

图3-16 SATA硬盘的电源
线

图3-17 SATA接口实物图

1. SATA规范的发展历程

SATA技术是Intel公司在IDF 2000大会上推出的，其最大的优势是传输速率高。SATA的工作原理非常简单：采用连续串行的方式来实现数据传输从而获得较高的传输速率。2003年发布的SATA 1.0规范提供的传输速率就已经达到了150MB/s，不但高出普通IDE硬盘所提供的100MB/s（ATA100），甚至超过了IDE最高传输速率133MB/s（ATA133）。

SATA在数据可靠性方面也有了大幅度提高。SATA可同时对指令及数据封包进行循环冗余校验（CRC），不仅可检测出所有单比特和双比特的错误，而且根据统计学的原理还能够检测出99.998%可能出现的错误。相比之下，PATA只能对来回传输的数据进行校验，而无法对指令进行校验，加上高频率下干扰甚大，因此数据传输稳定性很差。

除了传输速率更高、传输数据更可靠外，节省空间是SATA最具吸引力的地方。由于线缆相对于80芯的IDE线缆来说瘦了不少，更有利于机箱内部的散热，线缆间的串扰也得到了有效控制。不过SATA 1.0规范存在不少缺点，特别是缺乏对于服务器和网络存储应用所需的一些先进特性的支持。比如在多任务、多请求的典型服务器环境里面，SATA 1.0硬盘的确会有性能大幅度下降，还有可维护性不强、可连接性不好等缺点。这时，SATA 2.0的出现使这方面得到了很好的补充。

2. SATA 2.0规范中的新特性

与SATA 1.0规范相比，SATA 2.0规范中添加了一些新的特性，具体如下。

- **3Gb/s的传输速率：**在SATA 2.0扩展规范中，3Gb/s的速率是最大的亮点。由于SATA使用8bit/10bit编码，所以3Gb/s等同于300MB/s的接口速率。不过，从性能角度看，3Gb/s并不能带来多大的提升，即便是RAID应用的场合，性能提升也没有想象的那么大。因为硬盘内部传输速率还达不到与接口速率等同的程度。在大多数应用中，硬盘是将更多的时间花在了寻道上，而不是传输上。接口速率的提高直接影响的是从缓存进行读写的操作，所以理论上大缓存的产品会从3Gb/s的传输速率中得到更大的好处。

- 支持NCQ技术：在SATA 2.0扩展规范所带来的一系列新功能中，NCQ（Native Command Queuing，自身命令队列）功能也非常令人关注。硬盘是机电设备，容易受内部机械部件惯性的影响，其中旋转等待时间和寻道等待时间就大大限制了硬盘对数据访问和检索的效率。前面曾经描述过一个模型，指的就是这种由硬盘驱动器自身实现的排队技术。

如果对磁头寻道这个机械动作的执行过程实施智能化的内部管理，就可以大大地提高整个工作流程的效率。所谓智能化的内部管理就是取出队列中的命令，然后重新排序，以便有效地获取和发送主机请求的数据。在硬盘执行某一命令的同时，队列中可以加入新的命令并排在等待执行的作业中。如果新的命令恰好是处理起来机械效率最高的，那么它就是队列中要处理的下一个命令。但有效的排序算法既要考虑目标数据的线性位置，又要考虑其角度位置，并且还要对线性位置和角度位置进行优化，以使总线的服务时间最小，这个过程也称做“基于寻道和旋转优化的命令重新排序”。

台式SATA硬盘队列一直被严格地限制为深度不得超过32级。如果增加队列深度，可能会起到反作用——增加命令堆积的风险。通常SATA硬盘接收命令时有两种选择，一是立即执行命令，二是延迟执行。对于后一种情况，硬盘必须通过设置注意标志和Service位来通知主机何时开始执行命令。然而硬盘不能主动与主机通信，这就需要主机定期轮回查询，发现Service位后将发出一条Service命令，然后才能从硬盘处获得将执行哪一条待执行命令的信息。而且Service位不包含任何对即将执行命令的识别信息，所必需的命令识别信息是以标记值的形式与数据请求一同传输的，并仅供主机用于设置DMA引擎和接收数据缓冲区。这样主机就不能预先掌握硬盘所设置的辅助位是哪条命令设置的，

数据传输周期开始前也无法设置DMA引擎，这最终导致了SATA硬盘效率低下。

NCQ包含如下两部分内容。

一方面，硬盘本身必须有针对实体数据的扇区分布对命令缓冲区中的读写命令进行排序。同时硬盘内部队列中的命令可以随着必要的跟踪机制动态地重新调整或排序，其中跟踪机制用于掌握待执行和已完成作业的情况，而命令排队功能还可以使主机在设备对命令进行排队的时候，断开与硬盘间的连接以释放总线。一旦硬盘准备就绪，就重新连接到主机，尽可能以最快的速度传输数据，从而消除占用总线的现象。

另一方面，通信协议的支持也相当重要。因为以前的PATA硬盘在传输数据时很容易造成中断，这会降低主控器的效率，所以NCQ规范中定义了中断聚集机制。相当于一次执行完数个命令后，再对主控器回传执行完毕的信息，改善处理队列命令的效能。

从最早的希捷7200.7系列硬盘开始，NCQ技术应用于桌面产品的时间至今已超过半年，不过目前NCQ对个人桌面应用并没有带来多大的性能提升，在某些情况下还会引起副作用。而且不同硬盘厂商的NCQ方案存在着差异，带来的效果也不同。

- 端口选择器（Port Selector）：目前的SATA 2.0扩展规范还具备了Port Selector（端口选择器）功能。Port Selector 是一种数据冗余保护方案，使用Port Selector可增加冗余度，具有Port Selector功能的SATA硬盘，外部有两个SATA接口，同时连接这两个接口到控制器上，一旦某个接口坏掉或者连线故障，则立刻切换到另一个接口和连线上，不会影响数据传输。

- 端口复用器（Port Multiplier）：SATA 1.0的一个缺点就是可连接性不好，即连接多个硬盘的扩展性不好。因为在SATA 1.0规范中，一个SATA接口只能连接一个设备。SATA规范的制定者们显然也意识到了这个问题，于是在SATA 2.0中引入了Port Multiplier的概念。Port Multiplier是一种可以在一个控制器上扩展多个SATA设备的技术，它采用4位（bit）宽度的Port Multiplier端口字段，其中控制端口占用一个地址，因此最多能输出2的四次方减1个，即15个设备连接，这与并行SCSI相当。Port Multiplier的上行端口只有1个，在带宽为150MB/s的时候容易成为瓶颈，但如果上行端口支持300MB/s的带宽，就与Ultra320 SCSI的320MB/s十分接近了。Port Multiplier技术对需要多硬盘的用户很有用，不过目前提供这种功能的芯片组极少。
- 服务器特性：在SATA 2.0扩展规范中还增加了大量的新功能，比如防止开机时多硬盘同时启动带来太大电流负荷的交错启动功能；强大的温度控制、风扇控制和环境管理；背板互联和热插拔功能等。这些功能更侧重于低端服务器方面的扩展。
- 接口和连线的强化：作为一个还在不断添加内容的标准集合，SATA 2.0最新的热点是eSATA，即外置设备的SATA接口标准，采用了屏蔽性能更好的两米长连接线，目标是最终取代USB和IEEE 1394。在内部接口方面，Click Connect加强了连接的可靠性，在接上时有提示声，拔下时需要先按下卡口。这些细微的结构变化显示出SATA接口更加成熟和可靠。

下面单独用一节来讲解在存储方面应用最为广泛的SCSI硬盘接口。

3.5 SCSI硬盘接口

SCSI与ATA是目前现行的两大主机与外设通信的协议规范，而且它们各自都有自己的物理接口定义。对于ATA协议，对应的就是IDE接口；对于SCSI协议，对应的就是SCSI接口。凡是作为一个通信协议，就可以按照OSI模型（本书第7章将介绍）来将其划分层次，尽管有些层次可能是合并的或者是缺失的。划分了层次之后，我们就可以把这个协议进行分解，提取每个层次的功能和各个层次之间的接口，从而可以将这个协议融合到其他协议之中，形成一种“杂交”协议来适应各种不同的环境，这个话题将在本书第13章加以阐述。

SCSI的全称是Small Computer System Interface，即小型计算机系统接口，是一种较为特殊的接口总线，具备与多种类型的外设进行通信的能力，比如硬盘、CD-ROM、磁带机和扫描仪等。SCSI采用ASPI（高级SCSI编程接口）的标准软件接口使驱动器和计算机内部安装的SCSI适配器进行通信。SCSI接口是一种广泛应用于小型机上的高速数据传输技术。SCSI接口具有应用范围广、多任务、带宽大、CPU占用率低以及热插拔等优点。

SCSI接口为存储产品提供了强大、灵活的连接方式，还提供了很高的性能，可以有8个或更多（最多16个）的SCSI设备连接在一个SCSI通道上，其缺点是价格过于昂贵。SCSI接口的设备一般需要配合价格不菲的SCSI卡一起使用（如果主板上已经集成了SCSI控制器，则不需要额外的适配器），而且SCSI接口的设备在安装、设置时比较麻烦，所以远远不如IDE设备使用广泛。虽然从2007年开始，IDE硬盘就被SATA硬盘彻底逐出了市场。

在系统中应用SCSI必须要有专门的SCSI控制器，也就是一块SCSI控制卡，才能支持SCSI设备，这与IDE硬盘不同。在SCSI控制器上有一个相当于CPU的芯片，它对SCSI设备进行控制，能处理大部分的工作，减少了CPU的负担（CPU占用率）。在同时期的硬盘中，SCSI硬盘的转速、缓存容量、数据传输速率都要高于IDE硬盘，因此更多是应用于商业领域。

下面简单介绍一下SCSI规范的发展过程。

SCSI最早是1979年由美国的Shugart公司（希捷公司前身）制订的，在1986年获得了ANSI（美国标准协会）的承认，称为SASI（Shugart Associates System Interface），也就是最初版本SCSI-1。

SCSI-1是第一个SCSI标准，支持同步和异步SCSI外围设备；使用8位的通道宽度；最多允许连接7个设备；异步传输时的频率为3MB/s，同步传输时的频率为5MB/s；支持WORM外围设备。它采用25针接口，因此在连接到SCSI卡（SCSI卡上接口为50针）上时，必须要有一个内部的25针对50针的接口电缆。该种接口已基本被淘汰，在相当古老的设备上或个别扫描仪设备上可能还可以看到。

SCSI-2又被称为Fast SCSI，它在SCSI-1的基础上做了很大的改进，增加了可靠性，数据传输率也被提高到了10MB/s；但仍旧使用8位的并行数据传输，还是最多连接7个设备。后来又进行了改进，推出了支持16位并行数据传输的WIDE-SCSI-2（宽带）和FAST-WIDE-SCSI-2（快速宽带）。其中WIDE-SCSI-2的数据传输速率并没有提高，只是改用16位传输；而FAST-WIDE-SCSI-2则是把数据传输速率提高到了20MB/s。

SCSI-3标准版本是在1995年推出的，也习惯称为Ultra SCSI，其同

步数据传输速率为20MB/s。若使用16位传输的Wide模式时，数据传输率更可以提高至40MB/s。其允许接口电缆的最大长度为1.5米。

1997年推出了Ultra 2 SCSI (Fast-40) 标准版本，其数据通道宽度仍为8位，但其采用了LVD (Low Voltage Differential, 低电平微分) 传输模式，传输速率为40MB/s，允许接口电缆的最大长度为12米，大大增加了设备的灵活性，且支持同时挂接7个设备。随后推出了Wide Ultra 2 SCSI接口标准，它采用16位数据通道带宽，最高传输速率可达80MB/s，允许接口电缆的最大长度为12米，支持同时挂接15个装置。

LVD可以使用更低的电压，因此可以将差动驱动程序和接收程序集成到硬盘的板载SCSI控制器中。不再需要单独的高成本外部高电压差动组件。而老式SCSI需要使用独立的、耗电的高压器件。

LVD 硬盘可进行多模式转换。当所有条件都满足时，硬盘就工作在 LVD 模式下；反之，如果并非所有条件都满足，硬盘将降为单端工作模式。LVD硬盘带宽的增加对于服务器环境来说意味着更理想的性能。服务器环境都有快速响应、必须能够进行随机访问和大工作量的队列操作等要求。当使用诸如CAD、CAM、数字视频和各种RAID等软件的时候，带宽增加的效果能立竿见影，信息可以迅速而轻松地进行传输。

Ultra 160 SCSI，也称为Ultra 3 SCSI LVD，是一种比较成熟的SCSI接口标准，是在Ultra 2 SCSI的基础上发展起来的，采用了双转换时钟控制、循环冗余码校验和域名确认等新技术。在增强了可靠性和易管理性的同时，Ultra 160 SCSI的传输速率为Ultra 2 SCSI的2倍，达到160MB/s。这是采用了双转换时钟控制的结果。双转换时钟控制在不提高接口时钟频率的情况下使数据传输率提高了一倍，这也是Ultra 160

SCSI接口速率大幅提高的关键。

Ultra 320 SCSI，也称为Ultra 4 SCSI LVD，是比较新型的SCSI接口标准。Ultra 320 SCSI是在Ultra 160 SCSI的基础上发展起来的，Ultra 160 SCSI的3项关键技术，即双转换时钟控制、循环冗余码校验和域名确认，都得到了保留。以往的SCSI接口标准中，SCSI接口支持异步和同步两种传输模式。Ultra 320 SCSI引入了调步传输模式，在这种传输模式中简化了数据时钟逻辑，使Ultra 320 SCSI的高传输速率成为可能。Ultra 320 SCSI的传输速率可以达到320MB/s。

图3-18为SCSI总线连接示意图。

图3-18 SCSI总线连接示意图

下面介绍SCSI协议中的OSI模型。

上面描述的SCSI接口的各个规范，全部限于物理电气层，即描述传输速率、电气技术性能等。SCSI是一套完整的数据传输协议。一个通信协议必然会跨越OSI的所有7个层次，而物理电气参数只是OSI模型中的第一层，那么第二层到第七层，SCSI规范中也包含么？答案当然是肯定的。

1. SCSI协议的链路层

OSI模型中链路层的功能就是用来将数据帧成功地传送到这条线路的对端。SCSI协议中，利用CRC校验码来校验每个指令或者数据的帧，如果发现对方发来的校验码与本地计算的不同，则说明这个数据帧在传输过程中受到了比较强的干扰而使其中某个或者某些位发生了翻转，那么就会丢弃这个帧，发送方便会重传这个帧。

2. SCSI协议的网络层

1) SCSI总线编址机制

OSI模型中网络层的功能就是用来寻址的，那么面对总线或者交换架构下的多个节点，各个节点之间又是如何区分对方呢？只有解决了这个问题，才能继续，否则是没有意义的。SCSI协议利用了一个SCSI ID的概念来区分每个节点。在Ultra 320 SCSI协议中，一条SCSI总线上可以存在16个节点，其中SCSI控制器占用一个节点，SCSI ID被恒定设置为7。其他15个节点的SCSI ID可以随便设置但是不能重复。这16个ID中，7具有最高的优先级。也就是说，如果ID7要发起传输，则其他15个ID都必须乖乖把总线的使用权让给它。图3-19是SCSI总线ID优先级示意图。

图3-19 SCSI总线ID优先级示意图

由于总线是一种共享的线路，总线上的每个节点都会同时感知到这条线路上的电位信号，所以同一时刻只能由一个节点向这条总线上放数据，也就是给这条线路加一个高电位或者低电位。其他所有节点都能感知到这个电位的增减，但是只有接受方节点才会将感知到的电位增减信号保存到自己的缓存中，这些保存下来的信号就是数据。电路上是高电位则接受方会保存为1，低电位则保存成0，反过来保存也可以。图3-20所示的一个只有两条导线的总线网络（实际中的总线远远不止两条导线），其中总线终结电阻的作用是终结导线上的电位信号。

图3-20 SCSI总线ID

图3-21是一个32位数据总线的SCSI ID与其优先级以及导线的对应表。

图3-21 SCSI ID与其优先级以及导线的对应表

那么这些节点是如何知道现在正在通信的两个节点之中有没有自己呢？要了解当前线路上是不是自己在通信，或者自己想争夺线路的使用权而通告其他节点，这个过程叫做仲裁。有总线的地方就有仲裁，因为总线是共享的，各个节点都申请使用，所以必须有一个仲裁机制。SCSI接口并不只有8或者16条数据线，还有很多控制信号线。

普通台式机主板一般不集成SCSI控制器，如果想接入SCSI磁盘，则必须增加SCSI卡。SCSI卡一端接入主机的PCI总线，另一端用一个SCSI控制器接入SCSI总线。卡上有自己的CPU（频率很低，一般为RISC架构），通过执行ROM中的代码来控制整个SCSI卡的工作。经过这样的架构，SCSI卡将SCSI总线上的所有设备经过PCI总线传递给内存中运行着的SCSI卡的驱动程序，这样操作系统便会知道SCSI总线上的所有设备了。如果这块卡有不只一个SCSI控制器，则每个控制器都可以单独掌管一条SCSI总线，这就是多通道SCSI卡。通道越多，一张卡可接入的SCSI设备就越多。

图3-22是SCSI总线接入计算机总线的示意图。

图3-22 SCSI总线与计算机总线

2) SCSI寻址机制和几个阶段

（1）空闲阶段

总线一开始是处于一种空闲状态，没有节点要发起通信。总线空闲的时候，BSY和SEL这两条控制信号线的状态都为False状态（用一个持续的电位表示），此时任何节点都可以发起通信。

（2）仲裁阶段

节点是通过在8条或者16条数据总线上（8位宽SCSI有8条，16位宽SCSI有16条）提升自己对应的那条线路的电位来申请总线使用权的。提升自己ID对应线路的电位的同时，这个节点也提升BSY线路的电位。每个ID号对应这8条或者16条线中的一条。SCSI设备上都有跳线用来设置这个设备的ID号。跳线设置好之后，这个设备每次申请仲裁都只会在SCSI接口的8条或16条数据线中的对应它自身ID的那条线上提升电位。如果同时有多个节点提升了各自线路上的电位，那么所有发起申请的节点均判断总线上的这些信号，如果自己是最高优先级的，那么就持续保留这个信号。而其他低优先级的节点一旦检测到高优先级的ID线路上有信号，则立即撤销自身的信号，回到初始状态等待下轮仲裁，而最高优先级的ID就在这轮仲裁中获胜，取得总线的使用权，同时将SEL信号线提升电位。

SCSI总线的寻址方式，按照控制器—通道—SCSI ID—LUN ID来寻址。LUN是个新名词，全称是Logical Unit Number，后文会对它进行描述。

先看一下控制器一级寻址。控制器就是指SCSI控制器，这个控制器集成在南桥上，或者独立于某个PCI插卡。但不管在哪里，它们都要连接到主机IO总线上。有IO端口，就可以让CPU访问到。一个主机IO总线上不一定只有一个SCSI控制器，可以有多个，比如插入多张SCSI卡到主板，那么就会在Windows系统的设备管理器中发现多个SCSI控制器。系统会区分每个控制器。

每个控制器又可以有多个通道。通道也就是SCSI总线，一条SCSI总线就是一个通道。那么多条SCSI总线（通道）可以被一个控制器管理

么？答案是肯定的，这个物理控制器会被逻辑划分为多个虚拟的、可以管理多个通道（SCSI总线）的控制器，称为多通道控制器。目前市场上有的产品可以将4个通道集成到一个单独的SCSI卡上。不仅仅SCSI控制器可以有多通道，IDE控制器也有通道的概念。我们知道，普通台式机主板上一般会有两个IDE插槽，一个IDE插槽可以连接两个IDE设备，但是设备管理器中只有一个IDE控制器，如图3-23所示。也就是说一个控制器掌管着两个通道，每个通道（总线）上都可以接入两个IDE设备。

图3-23 Windows中的IDE控制器

常说的“单通道SCSI卡”和“双通道SCSI卡”，就是指上面可以接几条SCSI总线。当然通道数目越多，能接入的SCSI设备也就越多。

每个通道（总线）上可以接入8或16个SCSI设备，所以必须区分开每个SCSI设备。SCSI ID就是针对每个设备的编号，每个通道上的设备都有自己的ID。不同通道之间的设备ID可以相同，并不影响它们的区分，因为它们的通道号不同。如图3-24所示为多通道控制器示意图。

图3-24 多通道控制器示意图

图3-25中所示的机器安装了一块LSI的SCSI卡，但是显示为两个设备，这两个设备就是两个通道。

图3-25 Windows中的SCSI控制器

其中一个在第3号PCI总线上的第三个设备（第三个PCI插槽）上。功能0指的就是0号通道，如图3-26所示。

图3-26 Windows中的控制器通道号（1）

另一个也在第3号PCI总线上的第三个设备（第三个PCI插槽）上，表明这个设备也在同一块SCSI卡上。功能1指的是1号通道，如图3-27所示。

图3-27 Windows中的控制器通道号（2）

然而，SCSI ID并不是SCSI总线网络中的最后一层地址，还有一个LUN ID。这个是用来做什么的呢？难道一个SCSI ID，也就是一个SCSI设备，还可以再划分？是的，可以再分！再分就不是物理上的分割了，总不能把一个SCSI设备掰开两半吧。只能在逻辑上分，每个SCSI ID下面可以再区分出来若干个LUN ID。控制器初始化的时候，会对每个SCSI ID上的设备发出一条Report LUN指令，用来收集每个SCSI ID设备的LUN信息。这样，一条SCSI总线上可接入的最终逻辑存储单元数量就大大增加了。LUN对传统的SCSI总线来说意义不大，因为传统SCSI设备本身已经不可物理上再分了。如果一个物理设备上没有再次划分的逻辑单元，那么这个物理设备必须向控制器报告一个LUN0，代表物理设备本身。对于带RAID功能的SCSI接口磁盘阵列设备来说，由于会产生很多的虚拟磁盘，所以只靠SCSI ID是不够的，这时候就要用到LUN来扩充可寻址的范围，所以习惯上称磁盘阵列生成的虚拟磁盘为LUN。关于RAID和磁盘阵列会分别在本书的第4~6章中介绍。

（3）选择阶段

仲裁阶段之后，获胜的节点会将BSY和SEL信号线置位，然后将8或16条数据总线上对应它自身ID的线路和对应它要通信的目标ID的线路的电位提升，这样目的节点就能感知到它自己的线路上来了信号，开始做接收准备。

提示： SCSI控制器也是总线上的一个节点，它的优先级必须是最高的，即等于7，因为控制器需要掌控整条总线。

总线上最常发生的是控制器向其他节点发送和接收数据，而除控制器之外的其他节点之间交互数据，一般是不会发生的。如果要从总线上的硬盘复制数据到另一块硬盘，那也必须先将数据发送到控制器，控制器再复制到内存，经过CPU运算后再次发给控制器，然后控制器再发给另外一块硬盘。经过这么长的路径而不直接让这两块硬盘建立通信，原因就是硬盘本身是不能感知文件这个概念的，硬盘只理解SCSI语言，而SCSI语言是处理硬盘LBA块的，即告诉硬盘读或者写某些LBA地址上的扇区（块），而不可能告诉硬盘读写某个文件。文件这个层次的功能是由运行在主机上的文件系统代码所实现的，所以硬盘必须将数据先传送到主机内存由文件系统处理，然后再发向另外的硬盘。

这就是SCSI的网络层。每个节点都在有条不紊地和控制器交互着数据。

3. SCSI协议的传输层

OSI模型中的传输层的功能就是保障此端的数据成功地传送到彼端。与链路层不同的是，链路层只是保障线路两端数据的传送，而且一旦某个帧出错，链路层程序本身不会重新传送这个帧。所以，需要有一个端到端的机制来保障传输，这个机制是运行在通信双方最终的两端的，而不是某个链路的两端。

图3-28显示了SCSI协议是如何保障每个指令都被成功传送到对方的。

图3-28 控制器向设备发送数据（写入数据）

发起方在获得总线仲裁之后，会发送一个SCSI Command写命令帧，其中包含对应的LUN号以及LBA地址段。接收端接收后，就知道下一步对方就要传输数据了。接收方做好准备后，向发起方发送一个XFER_RDY帧，表示已经做好接收准备，可以随时发送数据。

发起方收到XFER_RDY帧之后，会立即发送数据。每发送一帧数据，接收方就回送一个XFER_RDY帧，表示上一帧成功收到并且无错误，可以立即发送下一帧，直到数据发送结束。

接收方发送一个RESPONSE帧来表示这条SCSI命令执行完毕。

图3-29是一个SCSI读过程的示意图。

图3-29 控制器向设备读取数据

发起方在获得总线仲裁之后，会发送一个SCSI Command读命令帧。接收端接收后，立即将该命令中给出的LUN以及LBA地址段的所有扇区的数据读出，传送给发起端。

所有数据传输结束后，目标端发送一个RESPONSE帧来表示这条SCSI命令执行完毕。

SCSI协议语言就是利用这种两端节点之间相互传送一些控制帧，来达到保障数据成功传输的目的。

4. SCSI协议的会话层、表示层和应用层

会话层、表示层和应用层是OSI模型中最上面的三层，是与底层网

络通信语言无关的，底层语言没有必要了解上层语言的含义。有没有会话层，完全取决于利用这个协议进行通信的应用程序。这里我们就不再详述了。

3.6 磁盘控制器、驱动器控制电路和磁盘控制器驱动程序

3.6.1 磁盘控制器

硬盘的接口包括物理接口，也就是硬盘接入到磁盘控制器上需要用的接口，具体的针数、某个针的作用等。除了物理接口规范之外，还定义了一套指令系统，叫做逻辑接口。磁盘通过物理线缆和接口连接到磁盘控制器之后，若想在磁盘上存放一个字母应该怎么操作？这是需要业界定义的很重要的东西。指令集定义了“怎样向磁盘发送数据和从磁盘读取数据以及怎样控制其他行为”，比如SCSI和ATA指令。其中，逻辑接口，也就是SCSI或者ATA指令集部分，指令实体内容是需要由运行于操作系统内核的驱动程序来生成的，而物理接口的连接，就是磁盘控制器芯片需要负责的，比如ATA控制器或SCSI控制器。磁盘控制器的作用是参与底层的总线初始化、仲裁等过程以及指令传输过程、指令传输状态机、重传、ACK确认等，将这些太过底层的机制过滤掉，从而向驱动程序提供一种简洁的接口。驱动程序只要将要读写的设备号、起始地址等信息，也就是指令描述块（Command Description Block, CDB）传递给控制器即可，控制器接受指令并做相应动作，将执行后的结果信号返回给驱动程序。

3.6.2 驱动器控制电路

应该将磁盘控制器和磁盘驱动器的控制电路区别开来，二者是作用于不同物理位置的。磁盘驱动器控制电路位于磁盘驱动器上，它专门负责直接驱动磁头臂做运动来读写数据；而主板上的磁盘控制器专门用来向磁盘驱动器的控制电路发送指令，从而控制磁盘驱动器读写数据。由

磁盘控制器对磁盘驱动器发出指令，进而操作磁盘，CPU做的仅仅是操作控制器就可以了。来梳理一下这个结构，CPU通过主板上的导线发送SCSI或者ATA指令（CDB）给同样处于主板上的磁盘控制器，磁盘控制器继而通过线缆将指令发送给磁盘驱动器并维护底层指令交互状态机，由磁盘驱动器解析收到的指令从而根据指令的要求来控制磁头臂。

SCSI或者ATA指令CDB是由OS内核的磁盘控制器驱动程序生成并发送的。CPU通过执行磁盘控制器驱动程序，生成指令发送给磁盘控制器，控制器收到这些CDB后，会做一定程度的翻译映射工作，生成最底层的磁盘可接受的纯SCSI指令，然后通过底层的物理操作，比如总线仲裁，然后编码，再在线缆上将指令发送给对应的磁盘。

3.6.3 磁盘控制器驱动程序

那么机器刚通电，操作系统还没有启动起来并加载磁盘控制器驱动的时候，此时是怎么访问磁盘的呢？CPU必须执行磁盘通道控制器驱动程序才能与控制器交互，才能读写数据。所以，系统BIOS中存放了初始化系统所必需的基本代码。系统BIOS初始化过程中有这么一步，就是去发现并执行磁盘控制器的Optional ROM（该ROM被保存在磁盘通道控制器中或者单独的Flash芯片内），该ROM内包含了该控制器的最原始的、可在主BIOS下执行的驱动程序，主BIOS载入并执行该ROM，从而就加载了其驱动程序，也就可以与控制器进行交互了。最后主BIOS通过执行驱动程序而使得CPU可以发送对应的读指令，提取磁盘的0磁道的第一个扇区中的代码载入内存执行，从而加载OS。

系统BIOS（主BIOS）中是包含常用的磁盘控制器驱动程序的，但是对于一些不太常用的较高端的板载控制器或者PCIE卡形式的控制卡，主BIOS一般不包含其驱动，所以必须主动加载其Optional ROM才能在

主BIOS下驱动。如果根本不需要在主BIOS下使用该控制器，那么就不必加载Optional ROM。在OS内核启动过程中，会用高性能的驱动程序来接管BIOS中驻留的驱动程序。当然，BIOS中也要包含键盘驱动，如果支持USB移动设备启动，还要有USB驱动。

图3-30显示了磁盘控制器驱动程序、磁盘控制器和磁盘驱动器控制电路三者之间的关系。控制器驱动程序负责将上层下发的SCSI/ATA指令传递给控制器硬件。

图3-30 磁盘控制器驱动程序、磁盘控制器和磁盘驱动器控制电路三者的关系

安装操作系统时，安装程序要求必须加载完整的磁盘控制器驱动程序之后才可以识别到控制器后面的磁盘从而才可以继续安装。此时虽然系统BIOS里的基本简化驱动已经可以向磁盘进行读写操作，但是其性能是很差的，基本都使用Int13调用方式；而现代操作系统都抛弃了这种方式，所以安装操作系统过程中必须加载完整驱动才可以获得较高的性能。至于系统安装完后的启动过程，一开始必须由BIOS来将磁盘的0磁道代码读出执行以便加载操作系统，使用的是简化驱动，启动过程中，OS的完整驱动会替代掉BIOS的简化驱动被加载。

提示： 本书第9章会详细描述SAN Boot的启动过程以及磁盘控制器驱动程序的详细架构。

3.7 内部传输速率和外部传输速率

3.7.1 内部传输速率

磁盘的内部传输速率指的是磁头读写磁盘时的最高速率。这个速率不包括寻道以及等待扇区旋转到磁头下所耗费时间的影响。它是一种理想情况，即假设磁头读写的时候不需要换道，也不专门读取某个扇区，而是只在一个磁道上连续地循环读写这个磁道的所有扇区，此时的速率就叫做硬盘的内部传输速率。

通常，每秒10000转的SCSI硬盘的内部传输速率的数量级大概在1000MB/s左右。但是为何实际使用硬盘的时候，比如复制一个文件，其传输速率充其量只是每秒几十兆字节呢？原因就是磁头需要不断换道。

想象： 闪电侠正在做数学题，假设我们不打断他，他每秒能连续做100道题，此时我们每隔0.1秒，就和他交谈打断他一次，每次交谈的时间是0.5秒。也就是说闪电侠实际上做数学题的时间是每隔0.5秒做一次，每次只能做0.1秒的时间。这样，每0.6秒闪电侠只能做10道题，那么可以计算出闪电侠实际每秒能做的数学题只有区区16道，这和我们不打断他时的每秒100道题相比差了6倍多。

同样，磁盘也是这个道理，我们不断地用换道来打断磁头。磁头滑过盘片一圈，只需要很短的时间，而换道所需的时间远远比盘片旋转一圈耗费的时间多，所以造成磁盘整体外部传输速率显著下降。有人问，必须要换道么？如果要读写的数据仅仅在一条磁道上，那是可以获得极

高的传输速率的，但是这并不容易实现。如今，随着硬盘容量的加大，应用程序产生的文件更是在肆无忌惮地加大，动辄几十、几百兆甚至上GB大小，敢问这种文件用一个磁道能放下么？显然不能。

所以，磁头必须不断地被“打断”去进行换道操作，整体传输速率就会大大降低。实际中一块10000转的SCSI硬盘的实际外部传输速率也只有80MB/s左右（最新的15000转的SAS硬盘外部传输率最大已经可以达到200MB/s）。为了避免磁头被不断打断的问题，人们发明了RAID技术，让一个硬盘的磁头在换道时，另一个磁盘的磁头在读写。如果有很多磁盘联合起来，同一时刻总有某块硬盘的磁头在读写状态而不是都在换道状态，这就相当于一个大虚拟磁盘的磁头总是处于读写状态，所以RAID可以显著提升传输速率。不仅如此，如果我们将RAID阵列再次进行联合，就能将速率在RAID提速的基础上，再次成倍地增加。这种工作，就需要大型磁盘阵列设备来做了。

提示： RAID技术的细节在本书第4章详细阐述；磁盘阵列技术将在本书第6章介绍。

3.7.2 外部传输速率

磁头从盘片上将数据读出，然后存放到硬盘驱动器电路板上的缓存芯片内，再将数据从缓存内取出，通过外部接口传送给主板上的硬盘控制器。从外部接口传送给硬盘控制器时的传输速率，就是硬盘的外部传输速率。这个动作是由硬盘的接口电路来发起和控制的。接口电路和磁头控制电路是不同的部分，磁头电路部分是超精密高成本的部件，可以保证磁头读写时的高速率。但是因为磁头要被不断地打断，所以外部接口传输速率无须和磁头传输速率一样，只要满足最终的实际速率即可。外部接口的速率通常大于实际使用中磁头读写数据的速率（计入换道的

损失)。

3.8 并行传输和串行传输

3.8.1 并行传输

来举一个例子，有8个数字从1到8，需要传送给对方。此时我们可以与对方连接8条线，每条线上传输一个字符，这就是并行传输。并行传输要求通信双方之间的距离足够短。因为如果距离很长，那么这8条线上的数字因为导线电阻不均衡以及其他各种原因的影响，最终到达对方的速度就会显现出差距，从而造成接收方必须等8条线上的所有数字都到达之后，才能发起下一轮传送。

并行传输应用到长距离的连接上就无优点可言了。首先，在长距离上使用多条线路要比使用一条单独线路昂贵；其次，长距离的传输要求较粗的导线，以便降低信号的衰减，这时要把它们捆到一条单独电缆里相当困难。IDE硬盘所使用的40或者80芯电缆就是典型的并行传输。40芯中有32芯是数据线，其他8芯是承载其他控制信号用的。所以，这种接口一次可以同时传输32b的数据，也就是4B。

提示：IO延迟与Queue Depth

IO延迟是指控制器将IO指令发出之后，直到IO完成的过程中所耗费的时间。目前业界有不成文的规定：只要IO延迟在20ms以内，此时IO的性能对于应用程序来说都是可以接受的，但是如果延迟大于20ms，应用程序的性能将会受到比较大的影响。我们可以推算出，存储设备应当满足的最低IOPS要求应该为 $1000/20=50$ ，即只要存储设备能够提供每秒50次IO，则就能够满足IO延迟小于等于20ms的要求。但是每秒50次，这显然太低估存储设备的能力了。

单块SATA硬盘能够提供最大两倍于这个最低标准的数值，而FC磁盘则可以达到4倍于这个数值。对于大型磁盘阵列设备，由若干磁盘共同接受IO，加上若干个IO通道并行工作，目前中高端设备达到十几万的IOPS已经不成问题。

然而，不能总以最低标准来要求存储设备。当接受IO很少的时候，IO延迟一般会很小，比如1ms甚至小于1ms。此时，每个IO通道的 $IOPS = 1000 / 1 = 1000$ ，这个数值显然也不对，上文所述的几十万IOPS，如果每个IO通道仅提供1000的IOPS，那么达到几十万，需要几百路IO通道，这显然不切实际。那么几十万的IOPS是怎么达到的呢？这就引出了另一个概念：Queue Depth。

控制器向存储设备发起的指令，不是一条一条顺序发送的，而是一批一批地发送，存储目标设备批量执行IO，然后将数据和结果返回控制器。也就是说，只要存储设备肚量和消化能力足够，在IO比较少的时候，处理一条指令和同时处理多条指令将会耗费几乎相同的时间。控制器所发出的批量指令的最大条数，由控制器上的Queue Depth决定。如果连接外部独立磁盘阵列，则一般主机控制器端可以将其Queue Depth设置为64、128等值，视情况而定。

如果给出Queue Depth、IOPS、IO延迟三者中的任意两者，则可以推算出第三者，公式为： $IOPS = (Queue\ Depth) / (IO\ Latency)$ 。实际上，随着Queue Depth的增加，IO延迟也会随即增加，二者是互相促进的关系，所以，随着IO数目的增多，将很快达到存储设备提供的最大IOPS处理能力，此时IO延迟将会陡峭地升高，而IOPS则增加缓慢。好的存储系统，其IO延迟的增加应该是越缓慢越好，也就是说存储设备内部应该具有快速IO消化能力。而对于消化不良的存储设备，其IO延迟将

升高得很快，以至于在IOPS较低时，IO延迟已经达到了20ms的可接受值。消化能力再高，也有饱和的时候。图3-31所示为IO延迟与Queue Depth示意图。

图3-31 IO延迟与Queue Depth示意图

3.8.2 串行传输

还是上面的例子，如果只用一条连线来连接到对方，则我们依次在这条线上发送这8个数字，需要发送8次才能将数字全部传送到对方。串行传输在效率上，显然比并行传输低得多。但是串行也有串行的优势，就是凭借这种优势使得硬盘的外部接口已经彻底被串行传输所占领。USB接口、IEEE 1394接口和COM接口，这些都是串行传输的计算机外部接口。

并行传输表面上看来比串行传输效率要高很多倍，但是并行传输有不可逾越的技术困难，那就是它的传输频率不可太高。由于在电路高速震荡的时候，数据线之间会产生很大的干扰，造成数据出错，所以必须增加屏蔽线。即使加了屏蔽线，也不能保证屏蔽掉更高的频率干扰。所以并行传输效率高但是速度慢。而串行传输则刚好相反，效率是最低的，每次只传输一位，但是它的速度非常高，现在已经可以达到10Gb/s的传输速率，但传输导线不能太多。

这样算来，串行传输反而比并行传输的总体速率更快。串行传输不仅仅用于远距离通信，现在就连PCI接口都转向了串行传输方式。PCIE接口就是典型的串行传输方式，其单条线路传输速率高达2.5Gb/s，还可以在每个接口上将多条线路并行，从而将速率翻倍，比如4X的PCIE最高可达16X，也就是说将16条2.5Gb/s的线路并行连接到对方。这仿佛又回到了并行时代，但是也只有在短距离传输上，比如主板上的各个部件

之间，才能承受如此高速的并行连接，远距离传输是达不到的。

3.9 磁盘的IOPS和传输带宽（吞吐量）

3.9.1 IOPS

磁盘的IOPS，也就是每秒能进行多少次IO，每次IO根据写入数据的大小，这个值也不是固定的。如果在不频繁换道的情况下，每次IO都写入很大的一块连续数据，则此时每秒所做的IO次数是比较低的；如果磁头频繁换道，每次写入数据还比较大的话，此时IOPS应该是这块硬盘的最低数值了；如果在不频繁换道的条件下，每次写入最小的数据块，比如512B，那么此时的IOPS将是最高值；如果使IO的payload长度为0，不包含开销，这样形成的IOPS则为理论最大极限值。IOPS随着上层应用的不同而有很大变化。

提示： 如何才算一次IO呢？这是很多人没有弄清楚的问题，也是定义很混乱的一个问题。其根本原因就在于一次IO在系统路径的每个层次上都有自己的定义。整个系统是由一个一个的层次模块组合而成的，每个模块之间都有各自的接口，而在接口间流动的数据就是IO。那么如何才算“一次”IO呢？以下列举了各个层次上的“一次”IO的定义。

应用程序向操作系统请求：“读取C:\read.txt到我的缓冲区”。操作系统读取后返回应用程序一个信号，这次IO就完成了。这就是应用程序做的一次IO。

文件系统向磁盘控制器驱动程序请求：“读取从LBA10000开始的后128个扇区”，然后“请读取从LBA50000开始的后64扇区”，这就是文件系统向下做的两次IO。这两次IO，假设对应了第一步里那个应用程序的请求。

磁盘控制器驱动程序用信号来驱动磁盘控制器向磁盘发送SCSI

指令和数据。对于SCSI协议来说，完成一次连续LBA地址扇区的读写就算一次IO。但是为了完成这次读或者写，可能需要发送若干条SCSI指令帧。从最底层来看，每次向磁盘发送一个SCSI帧，就算一次IO，这也是最细粒度的IO。但是通常说磁盘IO都是指完成整个一次SCSI读或者写。

如果在文件系统和磁盘之间再插入一层卷管理层，或在磁盘控制器和磁盘之间再插入一层RAID虚拟化层，那么上层的一次IO就往往会演变成下层的多次IO。

对于磁盘来说，每次IO就是指一次SCSI指令交互回合。一个回合中可能包含了若干SCSI指令，而这一个回合里却只能完成一次IO，比如“读取从LBA10000开始的后128个扇区”。

例如，写入10000个大小为1KB的文件到硬盘上，耗费的时间要比写入一个10MB大小的文件多得多，虽然数据总量都是10MB。因为写入10000个文件时，根据文件分布情况和大小情况，可能需要做好几万甚至十几万次IO才能完成。而写入一个10MB的大文件，如果这个文件在磁盘上是连续存放的，那么只需要几十个IO就可以完成。

对于写入10000个小文件的情况，因为每秒需要的IO非常高，所以此时如果用具有较高IOPS的磁盘，将会提速不少。而写入一个10MB文件的情况，就算用了有较高IOPS的硬盘来做，也不会有提升，因为只需要很少的IO就可以完成了，只有换用具有较大传输带宽的硬盘，才能体现出优势。

同一块磁盘在读写小块数据的时候速度是比较高的；而读写大块数据的时候速度比较低，因为读写花费的时间变长了。15000转的硬盘比10000转的硬盘性能要高。图3-32所示为磁盘IOPS性能与IO SIZE的关系

曲线。

图3-32 磁盘IOPS性能与IO SIZE的关系曲线

3.9.2 传输带宽

传输带宽指的是硬盘或设备在传输数据的时候数据流的速度。还是刚才那个例子，如果写入10000个1KB的文件需要10s，那么此时的传输带宽只能达到每秒1MB，而写入一个10MB的文件用了0.1s，那么此时的传输带宽就是100MB/s。所以，即使同一块硬盘在写入不同大小的数据时，表现出来的带宽也是不同的。具有高带宽规格的硬盘在传输大块连续数据时具有优势，而具有高IOPS的硬盘在传输小块不连续的数据时具有优势。

同样，对于一些磁盘阵列来说，也有这两个规格。一些高端产品同时具备较高的IOPS和带宽，这样就可以保证在任何应用下都能表现出高性能。

3.10 固态存储介质和固态硬盘

固态存储在这几年来开始大行其道，其在性能方面相对机械磁盘来讲有着无与伦比的优势，比如，没有机械寻道时间，对任何地址的访问耗费开销都相等，所以随机IO性能很好。关于SSD的一些性能指标，本节不再列出。

但是SSD也存在一些致命的缺点，现在我们就来了解一下固态存储。

提示： 关于固态硬盘的一些细节标准和操作指南请参考《固态硬盘火力全开——超高速SSD应用详解与技巧》，清华大学出版社。

3.10.1 SSD固态硬盘的硬件组成

SSD（Solid State Drive）是一种利用Flash芯片或者DRAM芯片作为数据永久存储的硬盘，这里不可再说磁盘了，因为Flash Drive不再使用磁技术来存储数据。利用DRAM作为永久存储介质的SSD，又可称为RAM-Disk，其内部使用SDRAM内存条来存储数据，所以在外部电源断开后，需要使用电池来维持DRAM中的数据。现在比较常见的SSD为基于Flash介质的SSD。所有类型的ROM（比如EPROM、EEPROM）和Flash芯片使用一种叫做“浮动门场效应晶体管”的晶体管来保存数据。每个这样的晶体管叫做一个“Cell”，即单元。有两种类型的Cell：第一种是Single Level Cell（SLC），每个Cell可以保存1bit的数据；第二种为Multi Level Cell（MLC），每个Cell可以保存2bit的数据。MLC容量为SLC的两倍，但是成本却与SLC大致相当，所以相同容量的SSD，MLC

芯片成本要比SLC芯片低很多。此外，MLC由于每个Cell可以存储4个状态，其复杂度比较高，所以出错率也很高。不管SLC还是MLC，都需要额外保存ECC校验信息来做数据的错误恢复。

我们来看一下这种场效应晶体管为何可以保存数据。由于计算机的数据只有0和1两种形式，那么只要让某种物质的存在状态只有两种，并且可以随时检测其状态，那么这种物质就可以存储1bit数据。磁盘使用一块磁粒子区域来保存1或者0，那么对于芯片来讲用什么来表示呢？当然非电荷莫属。比如充满电表示0，放电后表示1（这里指电子而不是正电荷）。浮动门场效应晶体管就是利用这种方法。

如图3-33所示，浮动门场效应管

图3-33 浮动门结构

由Controler Gate (CG)、Floating Gate (FG)、半导体二氧化硅绝缘层以及输入端源极和输出端汲极触点等逻辑元件组成。浮动门是一块氮氧化物，其四周被二氧化硅绝缘层包裹着，其外部为另一个门电路（即控制门）。在Word Line（字线）上抬高电势，会在S和D之间区域感生出一个电场，从而导通S和D、S和D之间有电流通过，这会使一部分电子穿过绝缘层到达浮动门内的氮氧化物，在这个充电过程中，电子电荷被存储在了浮动门中。随后Word Line恢复电势，控制门断开电场，但是此时电子仍然在被绝缘层包裹的浮动门电路中，所以此时浮动门被充电，并且这些电荷可以在外部电源消失之后依然依然可以保存一段时间。不同规格的Flash其保存时间不同，通常为几个月。这为系统设计带来了复杂性，Flash控制器必须确保每个Cell在电荷逐渐泄露到无法感知之前，恢复其原先的状态，也就是重新充电。而且要逐渐调整感知时间，由于对Cell的读操作是通过预充电然后放电来比对基准电压，进而判断1或0的，如果其中的电荷所剩不多，那么感知基准电压的变化就需要更长的等待时间，控制器需要精确

的做预判才可以保证性能。

所谓“浮动门”（Floating Gate, FG）的名称也就是由此而来的，即这块氮氧化物是被二氧化硅绝缘层包裹住而浮动在空中的，如图3-33右图所示，电流从Source极进入，从Drain极流出，浮动门一头与S极接触，一头与D极接触，由于被包裹有绝缘层，当电路电压达到一定阈值之后，可以被击穿导电，从而被充电或者放电。被充电到一定电势阈值的Cell的状态被表示为“0”，如果是MLC，可以保存4个电势状态，分别对应00、01、10、11，MLC能够用一个Cell存储两位，也是这个道理，要存哪个数值，就充电到哪个电势档位。FG上方有一片金属，称为控制门（CG），连接着字线，当需要用这个Cell表示“0”的时候，只需要在控制门上加一个足够高的正电压，导通S和D，产生电流，从而电子从Source-Drain电极穿越冲入到FG中。向FG中充电之后，Cell表示0，将FG放电，Cell表示1。

提示： 这有些难以记忆，因为按照常规思维，充电应当表示1而放电表示0，这里恰好相反。

至此我们了解到：Cell是利用FG中的电势值来与阈值对比从而判断其表示1或者0的。那么是否可以让一个Cell有多个阈值，让每一个阈值都表示一种状态呢？答案是可以。MLC类型的Cell就是这样做的。MLC模式的SSD，其每个Cell具有4个电势阈值，每次充电用特定电路掌握住火候，充到一定阈值之上但是低于下一个阈值，这样，利用4个阈值就可以表示4种状态了，4个阈值依次表示00、01、10、11。要向其写入什么值，只需要向其充多少火候的电就可以了。

如图3-33（物理图）和3-34（抽象图）为Cell阵列的有序排列图。我

图3-34 Cell阵列的有序排布图

们可以看到每个Cell串是由多个Cell串联而成的，每个Cell串每次只能读写其中一个Cell，多个Cell串并联则可以并行读写多位数据。通常一个Page中的所有位中的每个位均位于一个Cell串相同的位置上，那么对于一个使用2122B（含ECC）/Page的芯片来讲，就需要16896个Cell串，需要16896条串联导线。如图所示，将每个Cell串上所有Cell串联起来的导线称为“位线”，也就是串联每个Cell的S和D极的那根导线，同时也是电源线；将多个并联的Cell串中相同位置的Cell的CG金属片水平贯穿起来（并联）的那根导线组称为“字线”，这样就组成了一个二维Cell矩阵。

将多个这样的Cell排列在一起形成阵列，就可以同时操作多个比特了。NAND就是利用大量这种Cell有序排列而成的一种Flash芯片。如图3-35所示为一片16GB容量的Flash芯片的逻辑方框图。每 $4314 \times 8 = 34512$ 个Cell逻辑上形成一个Page，每个Page中可以存放4KB的内容和218B的ECC校验数据，Page也是Flash芯片IO的最小单位。每128个Page组成一个Block，每2048个Block组成一个区域（Plane），一整片Flash芯片由两个区域组成，一个区域存储奇数序号的Block，另一个则存储偶数序号的Block，两个Plane可以并行操作。Flash芯片的Page大小可以为2122B（含ECC）或者4313B（含ECC），一般单片容量较大的Flash其Page Size也大。相应地，Block Size也会根据单片容量的不同而不同，一般有32KB、64KB、128KB、512KB（不含ECC）等规格，视不同设计而定。

图3-35 Flash芯片逻辑图

如图3-36所示为Intel X-25M固态硬盘的拆机图，可以看到它使用了10片NAND Flash芯片，左上方为SSD控制器，左下方为RAM Buffer。最左侧为SATA物理接口。

图3-36 Cell阵列有序排布图

如图3-37所示为某SSD控制器芯片的方框图。其中包含多个逻辑模块，外围接口部分和底层供电部分我们就不去关心了。这里将目光集中在右半边，其中8051CPU通过将ROM中的Firmware载入IRAM中执行来实现SSD的数据IO和管理功能。Flash Controller负责向所有连接的NAND Flash芯片执行读写任务，每个NAND芯片用8b并行总线与Flash Controller上的每个通道连接，每时钟周期并行传递8b数据。Flash Controller与Flash芯片之间也是通过指令的方式来运作的，地址信息与数据信息都在这8位总线上传送，由于总线位宽太窄，所以一个简单的寻址操作就需要多个时钟周期才能传送完毕。芯片容量越大，那么地址也就越长，寻址时间也就越长，所以，对于小块随机IO，Flash会随着容量的增加而变得越来越低效。新的Flash芯片已经有16b总线的设计了。总线频率目前一般为33MHz，最新也有40MHz的。

图3-37 Flash控制芯片方框图

对于数据写入来说，待写入的数据必须经过ECC校验之后，将数据和ECC校验信息一并写入芯片；对于数据读取来说，数据会与其对应的ECC信息一起读出并作校验，校验正确后才会通过外部接口发送出去。ECC运算器位于Flash Controller中。整个SSD会有一片很大容量的RAM（相对于机械磁盘来讲），通常是64MB甚至128MB，其原因将在下文讲述。CPU执行的代码相对于机械磁盘来讲也是比较复杂的，关于CPU都需要执行哪些功能，也一并在下文讲述。

3.10.2 从Flash芯片读取数据的过程

MOS的导通并不是非通即断的，就算截止状态，也会有电流漏过，

只是非常弱而已。这里还要明确一点，向绝缘层内充电是指充入电子，充入负电荷，栅极电压越负，nMOS就越导不通，也就是说，漏电电流就越弱。如果不充电，反而漏电电流还高一些。

正是在这种前提下，从而可以检测出这种微弱电流的差别，用什么手段？还得SAMP上阵了，老生常谈，首先强制导通未选中的所有Cell的MOS，要读取的MOS栅极不加电压，然后给位线预充电（充正电荷，拉高电平），然后让位线自己漏电，如果对应的Cell里是充了电的（充的是电子负电荷），那么MOS截止性会加强（等效于开启电压升高），漏电很慢（电压相对维持在高位），如果没充电，则漏电很快（电压相对维持在低位），所以最终SAMP比较出这两种差别来，翻译成数字信号就是，充了电=电压下降的慢=电压比放了电的位线高=逻辑1，这么想你就错了。此处你忽略了一点，也就是SAMP不是去比对充了电的Cell位线和没充电的Cell位线，而是把每一根位线与一个参考电压比对，所以，这个参考电压一定要位于两个比对电压之间。具体过程是这样的，假设所有位线预充电结束时瞬间电压为1.0v，然后让位线自然放电（或者主动将位线一端接地放电）一段时间（非常短），在这段时间之后，原先被充了电的Cell其位线压降速度慢，可能到0.8v左右，而原先未被充电的Cell其位线压降速度较快，可能到0.4v左右，每一种Flash颗粒会根据大量测试之后，最终确定一个参考电压，比如0.6v，也就是位于0.4和0.8之间。那么当SAMP比对充电Cell位线时，参考电压小于位线电压，SAMP普遍都是按照参考电压>比对电压则为逻辑1，小于则为逻辑0的，所以最终的输出便是，充电Cell反而表示0，放了电的Cell反而表示1。这也正是NAND中的“N”（NOT，非）的来历，AND则是“与”，表示Cell的S和D是串接起来的，相当于串联的开关，它们之间当然是AND逻辑了。

如图3-38所示，当需要读出某个Page时，Flash Controller控制Flash芯片将相应这个Page的字线，也就是串接（实际上属于并联）同一个Page中所有Cell上的CG的那根导线，电势置为0，也就是不加电压，其他所有Page的字线的电势则升高到一个值，也就是加一个电压，而这个值又不至于把FG里的电子吸出来，之所以抬高电势，是为了让其他Page所有的Cell的S和D处于导通状态，而没被加电压的Cell（CG上的电势为0V），也就是我们要读取的那些Cell，其S和D的通断，完全取决于其FG中是否存有电子。

图3-38 读Page时的电压状态

SSD的IO最小单位为1个Page。所以，对于NAND Flash，通过“强行导通所有未被选中的Cell”AND“检测位线的通断状态”=“被选中的Cell的通断状态”，NOT“被选中的Cell的通断状态”=“位线的1/0值”。把整个Page的1或者0传输到芯片外部，放置于SSD的RAM Buffer中保存，这就完成了一个Page内容的读出。SSD的IO最小单位为1个Page。

3.10.3 向Flash芯片中写入数据的过程

对Flash芯片的写入有一些特殊的步骤。Flash芯片要求在修改一个Cell中的位的时候，在修改之前，必须先Erase（即擦除掉）这个Cell。我们暂且先不在此介绍为何要先Erase再修改的原因，先说一说这个“Erase”的意思，这里有点误导之意。我们以机械磁盘为例，机械磁盘上的“数据”是永远都抹不掉的，如果你认为将扇区全部写入0就算抹掉的话，那也是有问题的，你可以说它存放的全是数字0，这也是数据。那么SSD领域却给出了这个概念，是不是SSD中存在一种介于1和0之间的第三种状态呢，比如虚无状态？有人可能会联想一下，Cell“带负电”、“带正电”、“不带电”，这不就正好对应了3种状态么？我们可以

将“不带电”规定为“虚无”状态，是否将Cell从带电状态改为不带电状态就是所谓Erase呢？不是的。上文中叙述过，Cell带电表示0，不带电则表示1，Cell只能带负电荷，即电子，而不能带正电荷。所以Cell只有两种状态，而这两种状态都表示数据。

思考： 为何不以“带正电”表示1，“带负电”表示0，“不带电”表示中间状态？如果引入这种机制，那么势必会让电路设计更加复杂，电压值需要横跨正负两个域，对感应电路的设计也将变得复杂，感应电路不但需要感应“有无”还要感应“正负”，这将大大增加设计成本和器件数量。另外，感应电路的状态也只能有两种，即表示为0或者表示为1，如果0表示带负电，1表示带正电，那么就缺少一种用来表示不带电的电路状态，而这第三种状态在感应电路中是无法表示的，也就是说感应电路即使可以感知到三种状态但是也只能表示出两种状态。

其实，这里的Erase动作其实就是将一大片连续的Cell一下子全部放电，这一片连续的Cell就是一个Block。即每次Erase只能一下擦除一整个Block或者多个Block，将其中所有的Cell变为1状态。但是却不能单独擦除某个或者某段Page，或者单个或多个Cell。这一点是造成后面将要叙述的SSD的致命缺点的一个根本原因。Erase完成之后，Cell中全为1，此时可以向其中写入数据，如果遇到待写入某个Cell的数据位恰好为1的时候，那么对应这个Cell的电路不做任何动作，其结果依然是1；如果遇到待写入某个Cell的数据位为0的时候，则电路将对应Cell的字线电压提高到足以让电子穿过绝缘体的高度，这个电压被加到Control Gate上，然后使得FG从电源线（也就是位线上）汲取电子，从而对Cell中的FG进行充电，充电之后，Cell的状态从1变为0，完成了写入，这个写0的动作又叫做Programm，即对这个Cell进行了Programm。

图3-39 写入Cell时的电压状态

如图3-39所示，要写入某个Cell，首先也必须先选中其所在的Page，也就是将这个Page的字线加高电压，对应这个Cell的位线加0V电压，同一个串里的所有其他Page的字线也加一个高电压但是不如待写入Page的高，同时不需要写入操作的那些串对应的位线加一个对应字线相同的电压，结果就是，不需要写入数据的Cell的字线和位线电压抵消，电子不动；需要写入数据的Cell，也就是需要充电的Cell，由电势差将电子从位线中汲取过来充电。仔细看一下这个过程，就会发现，根本无法在这个二维矩阵中做到同时给一个Page里（一横行）的不同Cell既充电又放电，可以自己推演一下对应字线和位线的电压状态，你会发现永远做不到，是个矛盾，这也是Flash挥之不去的痛。我们可以思考一下，虽然不能够同时对一些cell有的写0、有的写1，那么是否可以把一个page的写入分成两个周期来做呢，比如第一个周期先把要写入0的所有Cell写0，第二个周期则要把写入1的所有Cell写1，或者反之？完全可以，但是设计者并没有选择这么做，因为对于Cell来讲，每个Cell都有一定的绝缘体击穿次数，比如MLC一个Cell可击穿一万次，那么就意味着，每个Page被写入1万次就会报废，这显然不能接受，有个办法就是让所有Cell循环写入，不写到原地，每次覆盖写要新写入其他位置，这样就能够保证所有Cell轮流被写入，平衡整体寿命。所以设计者选择了先预先准备好大片已经写满了全1的Cell，每次写入都写到这些预先备好的地方，只写0，如果是覆盖写，就需要把之前的Page地址做一个重定向，所以Flash控制器还需要保存一张地址重定向表。还有一个原因不用两个周期来实现对一个page的写入，那就是如果每次写入都要耗费2个周期，写性能便会骤降，所以设计者不得不采用一下子擦除大片的Block，也就是说不用写IO一次就放电一次，而是类似“批发”，然后写入的时候只需要1个周期即可，所付出的的代价就是后台需要不断地“批发”，一旦断链，那么性能便会骤降。

SSD会以Page为单位进行写入操作，写完一个Page，再写下一个Page。

Flash领域里，写又被称为Programm。由于Flash的最常见表现形式——EPRROM一般是只读的，但是一旦要更改其中的程序，则需要重新写入，即Re-Programm，所以就顺便将写入Flash的过程叫做Programm了。一块崭新的SSD上所有Cell都是已经被Erase好的，也可以使用特殊的程序对整个SSD重新整盘Erase。

提示： 为何字线并联了所有Cell的CG（也就是所有Cell共享CG的控制信号），而不是让每个Cell的CG可以被单独控制呢？这样做实际上还是为了成本和芯片面积考虑，技术上其实都可以实现，关键是钱的问题。

3.10.4 Flash芯片的通病

Flash芯片在写入数据的时候有诸多效率低下的地方。包括现在常用的U盘以及SSD中的Flash芯片，或者BIOS常用的EEPROM，它们都不可避免。

1. Flash芯片存储的通病之一：Erase Before Overwrite

对于机械磁盘来说，磁盘可以直接用磁头将对应的区域磁化成任何信号，如果之前保存的数据是1，新数据还是1，则磁头对1磁化，结果还是1；如果新数据是0，则磁头对1磁化，结果就变成了0。而Flash则不然，如果要向某个Block写入数据，则不管原来Block中是1还是0，新写入的数据是1还是0，必须先Erase整个Block为全1，然后才能向Block中写入新数据。这种额外的Erase操作大大增加了覆盖写的开销。

更难办的是，如果仅仅需要更改某个Block中的某个Page，那么此时就需要Erase整个Block，然后再写入这个Page。那么这个Block中除这个Page之外的其他Page中的数据在Erase之后岂不是都变成1了么？是的，所以，在Erase之前，需要将全部Block中的数据读入SSD的RAM Buffer，然后Erase整个Block，再将待写入的新Page中的数据在RAM中覆盖到Block中对应的Page，然后将整个更新后的Block写入Flash芯片中。可以看到，这种机制更加大了写开销，形成了大规模的写惩罚。这也是为何SSD的缓存通常很大的原因。

就像CDRW光盘一样，如果你只需要更改其上的几KB数据，那么就要先复制出全盘700MB的数据，然后擦除所有700MB，然后再写入更改了几KB数据的700MB数据。

SSD的这种写惩罚被称为Write Amplification（写扩大），我们依然使用写惩罚这个词。写惩罚有不同的惩罚倍数，比如，需要修改一个512KB的Block中的一个4KB的Page，此时的写惩罚倍数 $=512\text{KB}/4\text{KB}=128$ 。小块随机写IO会产生大倍数的写惩罚。

当SSD向Flash中的Free Space中写入数据时，并没有写惩罚，因为Free Space自从上次被整盘Erase后是没有发生任何写入动作的。这里又牵涉到一个比较有趣的问题，即存储介质如何知道哪里是Free Space，哪里是Occupied Space呢？本书中多个地方论述过这一点。只有文件系统知道存储介质中哪些数据是没用的，而哪些正在被文件系统所占用，这是绝对无可置疑的，除非文件系统通过某种途径通告存储介质。SSD也不例外，一块刚被全部Erase的SSD，其上所有Block对于文件系统或者SSD本身来讲，都可以认为是Free Space。随着数据不断的写入，SSD会将曾经被写入的块的位置记录下来，记录到一份Bitmap中，每一比特表示Flash中的一个Block。对于文件系统而言，删除文件

的过程并不是向这个文件对应的存储介质空间内覆盖写入全0或者1的过程，而只是对元数据的更改，所以只会更改元数据对应的存储介质区域，因此，删除文件的过程并没有为存储介质自身制造Free Space。所以说，对于SSD本身来讲，Free Space只会越来越少，最后导致没有Free Space，导致每个写动作都产生写惩罚，类似Copy On Write，而且Copy和Write很有可能都是一些在文件系统层已经被删除的数据，做了很多无用功，写性能急剧下降。对于一块使用非常久的SSD来讲，就算它在被挂载到文件系统之后，其上并没有检测到任何文件，文件系统层剩余空间为100%，这种情况下，对于SSD本身来讲，Free Space的比例很有可能却是0，也就是说只要曾经用到过多少，那么那个水位线就永远被标记在那里。

每个Block中的Page必须被按照一个方向写入，比如每个Block为128个Page，共512KB，则当这个Block被擦除之后，SSD控制器可以先向其中写入前32个Page（或者10个Page，数量不限），一段时间之后，可以再向这个Block中追加写入剩余的Page（或者多次追加一定数量的Page写入）而不需要再次擦除这个Block。SSD控制器会记录每个Block中的大段连续空余空间。但是不能够跳跃的追加，比如先写入0~31这32个Page，然后写入64~127这64个Page，中间空出了32个Page没有追加，控制器是不会使用这种方式写的，Page都是连续排布的。但是一般来讲，控制器都是尽量一次写满整个Block的从而可以避免很多额外开销。

2. Flash芯片存储的通病之二：Wear Off

随着FG充放电次数的增多，二氧化硅绝缘层的绝缘能力将遭到损耗，最后逐渐失去绝缘性，无法保证FG中保有足够的电荷。此时，这

个Cell就被宣判为损坏，即Wear Off。

损坏的Cell将拖累这个Cell所在的整个Page被标记为损坏，因为SSD寻址和IO的最小单位为Page。损坏的Page对应的逻辑地址将被重定向映射到其他完好的预留Page，SSD将这些重定向表保存在ROM中，每次加电均被载入RAM以供随时查询。

MLC由于器件复杂，其可擦写的寿命比较低，小于10000次。而SLC则高一些，十倍于MLC，小于100000次。这个值是很惊人的，对于某些场合下，有可能一天就可以废掉一大堆Cell/Page，几个月之内当预留的Page都被耗尽后，就会废掉整个SSD。这是绝对不能接受的。

写惩罚会大大加速Wear Off，因为写惩罚做了很多无用功，增加了不必要的擦写，这无疑使本来就很严峻的形势雪上加霜。但是对于读操作，理论上每个Cell可以承受高数量级的次数而不会损耗，所以对于读来说，无须担心。

3.10.5 NAND与NOR

3.10.2节已经讲过，这里再强调一下，当需要读出某个Page时，Flash Controller控制Flash芯片将相应这个Page的字线【也就是串连（实际上属于并联）同一个Page中所有Cell上的CG的那根导线】电势置为0，也就是不加电压，其他所有Page的字线的电势则升高到一个值，也就是加一个电压，而这个值又不至于把FG里的电子吸出来，之所以抬高电势，是为了让其他Page所有Cell的S和D处于导通状态，而没被加电压的Cell（CG上的电势为0V），也就是我们要读取的那些Cell，其S和D的通断完全取决于其FG中是否存有电子。说白了，未被选中的所有Cell，均强制导通，被选中的Cell的FG里有电，那么串联这一串Cell的

位线就会被导通，这是一种AND（也就是与）的关系；被选中Cell的FG里如果没电，那么其所处的Cell串的位线就不能导通（虽然串上的其他Cell均被强制导通），这也是AND的关系。也就是一串Cell必须全导通，其位线才能导通，有一个不导通，整条位线就不通。这就是NAND Flash中的AND的意义。那N表示什么？N表示Not，也就是非，NAND就是“非与”的意思。为什么要加个非？很简单，导通反而表示为0，因为只有FG中有电才导通，上文也说了，FG中有电反而表示为0，所以这就是“非”的意义所在。

还有一类NOR Flash，NOR就是“非或”的意思，大家自然会想到，位线一定不是串联的，而是并联的，才能够产生“或”的逻辑。实际上，在NOR Flash里，同样一串Cell，但是这串Cell中的每个Cell均引出独立的位线，然后并联接到一根总位线上；另外一点很重要的是，每个Cell的S和D之间虽然物理上是串连，但是电路上不再是串联，而是各自有各自的接地端，也就是每个Cell的S和D之间的通断不再取决于其他Cell里S和D的通断了，只取决于自己。以上两点共同组成了“或”的关系，同时每个Cell具有完全的独立性，此时只要通过控制对应的地线端，将未被选中的Cell地线全部断开，这样它们的S和D极之间永远无法导通（逻辑0状态），由于每个Cell的位线并联上联到总位线，总位线的信号只取决于选中的Cell的导通与否，对于被选中的Cell， $NOT \{ (“地线接通”AND“FG是否有电”) OR “未被选中Cell的输出” \} = “总位线的1/0值”$ ，这就是NOR非与门的逻辑。

由于NOR Flash多了很多导线，包括独立地线（通过地址译码器与Cell的地线相连）和多余的上联位线，导致面积增大。其优点是Cell独立寻址，可以直接用地址线寻址，读取效率比NAND要高，所以可以直接当做RAM用，但是由于擦除单位较小，擦除效率要比NAND低，所以

不利于写频繁的场景。

3.10.6 SSD给自己开的五剂良药

面对病入膏肓的SSD写入流程设计，是不是无可救药呢？好在SSD开出了五个药方。

1. 药方1：透支体力，拆东墙补西墙

为了避免同一个Cell被高频率擦写，SSD有这样一个办法：每次针对某个或者某段逻辑LBA地址的写都写到SSD中的Free Space中，即上一次全盘Erase后从未被写过的Block/Page中，这些Free Space已经被放电，直接写入即可，无须再做Copy On Write的操作了。如果再次遇到针对这个或者这段LBA地址的写操作，那么SSD会再次将待写入的数据重定向写到Free Space中，而将之前这个逻辑地址占用的Page标记为“Garbage”，可以回收再利用。等到Block中一定比例（大部分）的Page都被标记为“Garbage”时，并且存在大批满足条件的Block，SSD会批量回收这些Block，即执行Copy On Write过程，将尚未被标记为“Garbage”的Page复制到RAM Buffer，将所有Page汇集到一起，然后写入一个新Erase的Block，再将所有待回收的Block进行Erase操作，变成了Free Space。SSD这样做就是为了将写操作平衡到所有可能的Block中，降低单位时间内每个Block的擦写次数，从而延长Cell的寿命。

重定向写的设计可谓是一箭双雕，既解决了Wear Off过快问题，又解决了大倍数写惩罚问题（因为每次写都尽量重定向到Free Space中，无须CoW）。但是，正如上文所述，SSD自己认为的纯Free Space只会越来越少，那么重定向写的几率也就会越来越少，最后降至0，此时大倍数写惩罚无可避免。

由于Page的逻辑地址对应的物理地址是不断被重定向的，所以SSD内部需要维护一个地址映射表。可以看到这种设计是比较复杂的，需要SSD上的CPU具有一定的能力运行对应的算法程序。这种避免Wear Off过快的重定向算法称为Wear Leveling，即损耗平衡算法。

Wear Leveling的实现方法随不同厂商而不同，有些以一块大区域为一个平衡范围，有些则完全顺序地写完整个SSD的Free空间，然后再回来顺序地写完整个被回收的Free空间，无限循环直到Free空间为0为止。传统机械硬盘中，逻辑上连续LBA地址同样也是大范围物理连续的，但是对于SSD，逻辑和物理的映射随着使用时间的增长而越来越乱，好在SSD不需要机械寻址，映射关系乱只会影响CPU计算出结果的时间而不会影响数据IO的速度，而CPU运算所耗费的时间与数据IO的时间相比可以忽略不计，所以映射关系再怎么乱也不会对IO的性能有多少影响。

利用这种方式，SSD内部实现了垃圾回收清理以及新陈代谢，使得新擦除的Block源源不断地被准备好从而供应写操作。

有必要一提的是，Flash控制器的这种机制又可以被称为RoW，也就是Redirect On Write，每遇到需要更新的页面，Flash控制器便将其缓存在RAM中，当缓冲的待写入页面达到了一个Block容量的时候，便会直接将这页面写入到一个已经擦好的Block中。如果待写入的页面未攒够一个Block的容量，必要时也可以写入一个擦好的Block，此时这个Block处于未写满状态，随后可以继续写入页面直到写满为止，当然，要做到这一点，Flash控制器就需要为每个Block记录断点信息了。

每次被更新的页面在更新之前所处的位置，会在该页面被重定向写入到其他Block之后，在映射表中标记为垃圾，可以想象，随着使用时间的加长，Flash中Block里的这种垃圾孔洞越来越多，而且越来越不连

续，到处都是，可谓是千疮百孔。垃圾回收程序最喜欢的就是一个Block里全是垃圾的状态，此时最好，但是如果多数Block都处于一种“不尴不尬”的状态，比如50%内容是垃圾，但是另外50%的内容却未被标记为垃圾，那么此时到底是否回收？要回收这50%的垃圾，就需要先把那些非垃圾内容读出到RAM中存放，随后和新数据一起一视同仁地写入到擦好的Block中，此时产生50%不必要的读和50%不必要的写操作，这些都属于惩罚操作，越是不尴不尬，后台的惩罚就越多，性能就会越差。

2. 药方2：定期清除体内垃圾，轻装上阵

通过上面的论述我们知道，影响一块SSD寿命和写入性能的最终决定因素就是Free Space，而且是存储介质自身所看到的Free Space而不是文件系统级别的Free Space。但是SSD自身所认识的Free Space永远只会少于文件系统的Free Space，并且只会越来越趋于0。所以，要保持SSD认识到自身更多的Free Space，就必须让文件系统来通知SSD，告诉它哪些逻辑地址现在并未被任何文件或者元数据所占用，可以被擦除。这种思想已经被实现了。所有SSD厂商均会提供一个工具，称为“Wiper”，在操作系统中运行这个工具时，此工具扫描文件系统内不用的逻辑地址，并将这些地址通知给SSD，SSD便可以将对应的Block做擦除并回收到Free Space空间内。如果用户曾经向SSD中写满了文件随后又删除了这些文件，那么请务必运行Wiper来让SSD回收这些垃圾空间，否则就会遭遇到大写惩罚。

Wiper并不是实时通知SSD的，这个工具只是一次性清理垃圾，清理完后可以再次手动清理。所以，这个工具需要手动或者设置成计划任务等每隔一段时间执行一次。

这种垃圾回收与上文中的那种内部垃圾回收不在一个层面上，上文中所讲的是SSD内部自身的重定向管理所产生的垃圾，而本节中所述的则是文件系统层面可感知的垃圾，被映射到SSD内部，也就变成了垃圾。

3. 药方3：扶正固本，调节新陈代谢使其持续清除体内垃圾

定期执行垃圾清理确实可以解燃眉之急，但是有没有一种方法，可以让文件系统在删除某个文件之后实时地通知SSD回收对应的空间呢？这种方法是有的。TRIM便是这种方法的一个实现。TRIM是ATA指令标准中的一个功能指令，在Linux Kernal 2.6.28中已经囊括，但是并不完善。Windows 7以及Windows Server 2008R2中已经提供了完善的TRIM支持。一些较早出现的SSD也可以通过升级Firmware来获得对TRIM的支持。

TRIM可以使SSD起死回生，经过实际测试，开启了TRIM支持的SSD，在操作系统TRIM的支持下，可以成功地将性能提高到相对于SSD初始化使用时候的95%以上，写惩罚倍数维持在1.1倍左右。

提示：台湾闪存厂商PhotoFast银箭已经发布了带有自行回收垃圾空间的SSD，不依赖于Trim指令，在此可以推断其SSD内部一定被植入了可识别NTFS格式数据的代码，所以可以自行识别NTFS文件系统中不被文件占用的空间。

4. 药方4：精神修炼法，提升内功

Delay Write是一种存储系统常用的写IO优化措施。比如有先后两个针对同一个地址的IO——Write1、Write2，先后被控制器收到，而在

Write1尚未被写入永久存储介质之前，恰好Write2进入，此时控制器就可以直接在内存中将Write2覆盖Write1，在写入硬盘的时候只需要写入一次即可。这种机制为“写命中”的一种情况（其他情况见本书后面章节）。它减少了不必要的写盘过程，对于SSD来讲，这是很划算的。

然而，如果一旦遇到这种IO顺序比如Write1、Read2、Write3，如此时控制器先将Write3覆盖到Write1，然后再处理Read2的话，那么Read2原本是应该读出Write1的内容的，经过Delay Write覆盖之后，却读出了Write3的内容，这就造成了数据不一致。

所以，控制器在处理Delay Write时要非常小心，一定要检测两个针对同一个地址的写IO之间是否插有针对同一个地址的读IO，如果有读IO，首先处理读，然后再覆盖。

Combine Write是另一种存储系统控制器常用的写IO优化方法。对于基于机械硬盘的存储系统，如果控制器在一段时间内收到了多个写IO而这些写IO的地址在逻辑上是连续的，则可以将这些小的写IO合并为针对整体连续地址段的一个大的IO，一次性写入对应的磁盘，节约了很多SCSI指令周期，提高了效率。对于SSD来讲，由于SSD中的逻辑地址本来就是被杂乱地映射到可能不连续的物理地址上的，但是并不影响多少性能，所以，SSD控制器可以整合任何地址的小块写IO成一个大的写IO而不必在乎小块写IO针对的逻辑地址是否连续。整合之后的大写IO被直接写向一个Free的Block中，这样做大大提高了写效率。

5. 药方5：救命稻草，有备无患

为了防止文件系统将数据写满的极端情况，SSD干脆自己预留一部分备用空间用于重定向写。这部分空间并不通告给操作系统，只有SSD

自己知道，也就是说文件系统永远也写不满SSD的全部实际物理空间，这样，就有了一个永远不会被占用的一份定额的Free Space用于重定向写。Intel X25-E系列企业级SSD拥有20%的多余空间。其他普通SSD拥有6%~7%的比例。

思考： 为何用普通碎片整理程序在文件系统层整理碎片对于SSD来说是雪上加霜，理解这个问题的关键在于理解对于SSD来讲，逻辑上连续的地址不一定在物理上也连续，如果使用普通碎片整理程序，不但不能达到效果，反而还会因为做了大量无用功而大大减少SSD的寿命。

因为SSD需要对数据进行合并以及其他优化处理以适应Flash的这些劣势，所以SSD自身对接收的写IO数据使用Write Back模式，即接收到主机控制器的数据后立即返回成功，然后异步的后台处理和刷盘。所以，一旦遇到突然掉电，那么这些数据将会丢失，正因如此，SSD需要一种掉电保护机制，一般是使用一个超级电容来维持掉电之后的脏数据刷盘。

关于SSD性能方面的更详细的内容请参考本书第19章中的部分内容。

3.10.7 SSD如何处理Cell损坏

对于机械硬盘，如果出现被划伤的磁道或者损坏的扇区，也就意味着对应的磁道或者扇区中的磁粉出现问题，不能够被成功地磁化，那么磁头会感知到这个结果，因为磁化成功必定对应着电流的扰动，如果针对某块磁粉区磁化不成功，磁头控制电路迟迟没感知到电流扰动，或者扰动没有达到一定程度，那么就证明这片区域已经损坏。而对于Flash中

的Cell，当Cell中的绝缘体被击穿一定次数（SLC 10万次，MLC 1万次）之后，损坏的几率会变得很高，有时候不见得非要到这个门限值，可能出厂就有一定量损坏的Cell，使用一段时间之后也可能时不时出现损坏。那么SSD如何判断某个Cell损坏了呢？我们知道Cell损坏之后的表现是只能表示1而无法再被充电并且屏蔽住电子了，如果某个Cell之前被充了电，为0，某时刻Cell损坏，漏电了，变为1，那么在读取这个Cell的值的时候，电路并不会感知到这个Cell之前的值其实是0，电路依然读出的是1，那么此时问题就出现了。解决这个问题的办法是使用ECC纠错码，每次读出某个Page之后，都需要进行ECC校验来纠错。每种Flash生产厂商都会在其Datasheet中给出一个最低要求，即使用该种颗粒起码需要配合使用何种力度的纠错码，比如8b@512B或者24b@1KB等。8b@512B意味着如果在512字节范围内出现8b的错误，则是可以纠错恢复的，如果超过了8b，那么就无法纠错了，此时只能向上层报“不可恢复错误”；同理，24b@1KB也是一样的意思。厂商给出的纠错码力度越低，就证明这种Flash颗粒的品质越好，损坏率越低。

3.10.8 SSD的前景

以上一切缓解SSD效率问题的方法，都是治标不治本。随着Cell的不断损坏，最后的救命稻草——SSD私自保留的空间也将被耗尽，没了救命稻草，加之文件系统空间已满的话，那么SSD效率就会大大降低。

但是，我们在这里讨论的SSD写效率降低，不是与机械硬盘相比的。瘦死的骆驼比马大，SSD比起机械硬盘来讲其优势还是不在一个数量级上的。

可以将多块SSD组成Raid阵列来达到更高的性能。但是可惜的是，Raid卡目前尚未支持TRIM。

为了适应Flash的存储方式，有多种Flash Aware FS被开发出来，这些Flash FS包括TrueFFS、ExtremeFFS等。这些能够感知Flash存储方式的FS，可以将大部分SSD内部所执行的逻辑拿到FS层来实现，这样就可以直接在上层解决很多问题。

SSD看似风光无限，但是其技术壁垒比较大，为了解决所产生的多个问题，设计了多种补救措施，需要靠TRIM维持，而且数据不能占空间太满，否则无药可救。基于目前NAND Flash的SSD很有可能是昙花一现。目前市场上已经出现SRAM、RRAM等更快的永久存储介质，随着科技的发展，更多更优良的存储介质和存储方式必将替代机械硬盘的磁碟，SSD数据存储次世代即将到来。

此外，SSD在使用的时候也比较尴尬，SSD的成本还是太高，比如用户需要一个10TB的存储系统，不可能都用SSD，此时怎么规划？对某个Raid组专门使用SSD来构建，比如8个256GB的SSD组成某Raid组，有效容量1TB，再在其上划分若干LUN。用户使用一段时间后发现原来人工预测的热点数据已经冷了，而原来的冷数据热了，那么此时需要数据迁移，这就要兴师动众了。或者用户发现某个应用所需的数据量庞大，但是又不可能将其分割开一部分放到SSD中一部分放到传统磁盘中。这些尴尬的境况都限制了SSD的应用，也是多个厂商相继开发了Automatically Storage Tiering技术的原因（见后面章节）。

针对传统SSD使用时候的尴尬境况，Seagate等厂商也相继推出了SSD+HDD混合式存储硬盘。这种硬盘其实是将Flash芯片用于磁盘的二级缓存，一级缓存是Ram，二级是Flash，三级则是磁盘片。利用这种多级缓存更大程度地降低了磁头寻道的影响。有人质疑说，传统磁盘自身已经有了多达64MB的RAM缓存，还需要Flash再作为下一级的缓存么？是的，64MB的容量还是太小，虽然速度高，但是很快就被充满，同时

没有掉电保护机制，所以盘阵控制器不会让磁盘以WriteBack模式操作，充满了则一定要连续地刷到盘片中，这样就不可避免地直接导致可见的性能骤降；而使用Flash芯片再加入一级缓存，比如用8GB甚至16GB的Flash芯片，数据从RAM缓存出来后先被存入Flash，不用寻道，同时可以掉电保护，Write Back之后，磁盘驱动器再在后台将数据从Flash中写入磁盘片保存并清空Flash中的内容。这样的做法，既比传统纯SSD便宜，还保证了性能，有效屏蔽了磁头寻道带来的高延迟，同时又保证了容量（数据后台被刷入磁盘片），传统SSD在容量方面的问题就这么被解决了。这种磁盘短期内尚无法取代传统纯SSD，因为其读取操作依然需要从磁盘片中读取几乎所有数据而不是从Flash中，无法与纯SSD抗衡，但是其取代传统磁盘的趋势是存在的，今后的磁盘如果都这么做，会大大增加单个磁盘的随机写性能，对磁盘阵列控制器的更新与设计也是一个挑战。

提示： 试想一下，磁盘阵列控制器作为一个嵌在主机与后端物理磁盘之间的角色，其一个本质作用就是提升性能，其性能提升的原理就是让众多磁盘同时工作以屏蔽磁头寻道所浪费的时隙，而现在，每块磁盘自身就能够利用Flash芯片来屏蔽寻道的时隙，如何办？充分地想象吧。

3.11 Memblaze闪存产品介绍

机械磁盘由于其高复杂度的机械部件、芯片及固件，让人望而却步，其技术只被掌握在少数几家巨头手中。因为初期成本非常高，如果没有销量，那就是赔本。而近年来出现越来越多的国产企业级固态存储产品，SSD相对机械磁盘为何会有这么低的门槛？那是必须的。组成SSD的主要是Flash颗粒和控制器，而这些部件，都可以从各种渠道购买，当然，品质和规格也是参差不齐，但是，SSD是对各种部件的集成，只要有足够的集成能力，能快速看懂这些部件提供商的手册，对整个存储系统有基本了解，那么只要具有一定水平和经验的硬件研发者，都可以进入这个领域。

但是，目前各种固态介质充斥市场，从TF卡、U盘到消费级SATA SSD，再到企业级SATA SSD，再到企业级PCIE闪存卡，每个档次要求很不一样。

要做成企业级SSD，不但需要有技术能力，还得有充分的产品定制化能力。我们很欣喜地看到Memblaze同时具有了这两种能力。

3.11.1 技术能力

评判一个存储团队的技术能力，第一看他们是贴牌还是有自己的特色；第二看其产品设计采用硬加速还是软加速。说到软加速，也就是把原本由硬件芯片实现的功能，上提到由主机CPU运行代码来实现，也就是常说的“软件定义”概念里的一个意思。个人感觉软件定义是个好事情，但是不能过分定义，否则会适得其反。比如一些硬加速DSP、底层特定协议编解码器，这些如果用通用CPU来实现也可以，只要你能忍受

性能的极度下降以及几近100%的CPU负荷。曾经有某国外厂商的PCIE闪存卡使用的就是软加速方案。

我们首先得了解一下对于Flash闪存卡，其控制器都需要做些什么工作，然后才能判断到底是软加速就够了，还是必须用硬加速。通过本章前文的描述，我们已经充分了解了Flash的读写过程，也深知其复杂性，就因为不能够对同一个Page里的不同Cell同时放电和充电，导致了后续一系列严重后果，此外，Flash Cell单元的低寿命和高出错率，也是个令人头疼的事。处理所有这堆烂摊子的角色，其学术名词叫做FTL（Flash Translation Layer），其意思一个是把Flash基于页面为最小IO单元映射成传统的块设备以512B扇区为最小IO单位，另一个是把逻辑IO地址映射为物理IO地址，因为每个Page的实际物理地址都会不断变化。当然，这只是地址方面的映射处理，FTL需要同时掌管映射处理和对Flash的磨损均衡、垃圾回收、纠错等等处理。

我们继续分析这套架构都需要哪些数据结构。首先需要多张超大二维表来存储逻辑地址、物理地址的映射记录，后端挂接的Flash颗粒容量越大，这些表（数组）就越大；其次是为了加速查询所做的索引、位图之类的元数据结构；还有就是存储IO栈里常用的一种数据结构——链表，或者称其为“描述体”，用来追踪分散在多个物理内存处但是逻辑上是一个整体的事物，比如逻辑连续空间，或者某个IO任务等；另外，状态机也非常复杂。

我们分析一下，上面这套数据结构和状态机，如果完全将其运行在主机OS底层的设备驱动层，或者干脆运行在用户态（一般都会是在用户态运行，因为驱动层不适合做复杂的逻辑处理），效率究竟有多高。这一点想想就可以大致推断，比如TCPIP协议栈是个纯软协议栈，其不需要维护太多元数据，多数计算量位于状态机的判断和输出上，即便是

这样，一个万兆网口之上承载iSCSI协议的场景，在IOPS跑满之后，CPU利用率基本都超过50%了，何况是需要查表映射的场景。另外，软加速方案需要设备传输更多的信息给驱动程序，这直接导致中断次数激增。还有，软加速方案需要在主机端保存大量的元数据，对内存的占用是不可小视的，通常1GB起步，量多加价，而硬加速方案只有主机端驱动运行耗费一部分内存，通常1MB起步。

如果是使用硬加速方案，所有的元数据、状态机，都在Flash控制器内部维护和计算，就能大大降低主机端CPU的负荷，主机端CPU的负荷只体现在响应外部设备的中断上了。硬加速的好处就是可以将一些专用逻辑直接做成硬逻辑电路，在一个或者几个时钟周期内就可以完成通用CPU需要耗费几百个或者几千个时钟周期才能完成的任务。举个例子，对ECC码的计算，就可以单独拿出来做成硬加速电路。有些逻辑不能完全被翻译成纯数字电路，比如地址映射查表等，一小部分逻辑必须依靠通用CPU来运行对应的固件（或者说微码）来协调完成全部逻辑，当然那些重复运算还是交给数字电路，微码只是负责协调和总控，这一点如果是一些比较复杂的逻辑的话，靠纯数字电路是无法完成的。这里的“通用CPU”并非指主机端CPU，而是指控制器内部的嵌入式CPU核心，一个控制其芯片内部集成的通用CPU核心数量可能在几个到十几个这种级别。

我们可以看到，如果用硬加速方案，对设计者的要求，不仅仅是了解Flash FTL层的全部逻辑，还得有充分的技术实力把这些逻辑梳理成数据结构，然后判断哪些逻辑可以被硬加速，然后还得具备将软件翻译成硬件逻辑的技能，除此之外，还得具有熟练驾驭FPGA的技能，因为不可能让建筑师去烧窑制造砖头，砖头肯定是要买，FPGA也得用现成的。

另外，一款固态存储产品，选择使用什么样的IO协议也是至关重要的。对于基于ATA协议的SATA接口来讲，是无法发挥出Flash最优性能的，直接使用PCIE接口接入主机总线是目前离CPU最近的途径，但是这样就无法使用SATA协议的一切已有软硬件了，包括成熟的SATA控制器硬件，以及主机端OS系统内核对SATA的原生驱动支持。选择PCIE就意味着必须自己开发一套轻量级IO访问协议。

提示： 目前针对PCIE Flash的专用IO协议有NVMe和SCSIe两种，现在看来NVMe似乎已经占了上风，因为SCSI协议簇的庞大臃肿已经完全不适合Flash这种高速介质了。Memblaze在第一代产品中并没有使用NVMe，而是使用了自己的私有协议，下一代产品很有可能会过渡到NVMe标准上来。

Memblaze很显然是充分掌握了这条线上的所有技能。图3-40是Memblaze的Flash控制器框图。其使用了成品FPGA，其中IP硬核部分为PCIE控制器及DDR3控制器；IP软核部分为DMA控制器、嵌入式通用CPU等等，可以任意生成，其他都是需要用户自定义设计的部分。其中纯硬逻辑包含Flash通道控制器、ECC计算电路等；需要嵌入式通用CPU辅助的有：地址映射器、磨损均衡和垃圾回收模块以及整个控制器的中央协调控制逻辑处理部分等。

图3-40 Memblaze的Flash控制器框图

除了对硬件的掌控能力之外，Memblaze还拥有IO延时平滑的专利技术，可以针对个别超长延时的IO请求进行削峰滤波处理（类似电容器滤除高电压脉冲的工作原理，对IO延时进行滤波），当运行在较高IOPS情况下，控制器会自动调整垃圾回收算法和内部等待队列深度，并将IO延时进行平滑处理，从而避免产生超长延时的IO，减少对后端系统

的影响，使得用户的应用运行得更加平稳顺滑。图3-41为实测结果，可以看到抖动很少，有些场景根本没有抖动。这项技术主要是采用排队论和现代控制理论对SSD的一些指标进行采样，根据采样结果去控制系统通路上的参数，用来优化IO的抖动和延迟。比如，用户发给SSD的请求可能是顺序写入，也可能是随机写入。不同的写入模式对应的写放大倍数是不同的。在这种情况下，后端的处理速度在不同的输入下就会有不同的通道阻塞程度，对于IO来说就会造成抖动和服务质量较低。如果采用通道的阻塞程度作为控制变量，运用自动控制理论，就可以动态地均衡前端的压力，降低整个系统的延迟和抖动。

图3-41 Memblaze的时延平滑技术

最后，对于固态存储厂商来讲，能够驾驭各Flash颗粒厂商的NAND Flash也显得至关重要。不同颗粒使用不同的规格，比如ECC位比例、页面大小、访问协议等等。这些都需要花上足够的时间去测试、调优。

提示： 本文落笔时，Memblaze的最终产品性能已经可以达到70w+的IOPS，已经是目前PCIe闪存卡的最高纪录。

另外，NAND Flash的写放大是导致性能和寿命下降的主要原因，是否能够充分降低写放大效应，也是体现技术实力的地方。如图3-42所示，Memblaze对写放大的压制还有很不错的。

图3-42 Memblaze对写放大的压制

3.11.2 产品能力

技术是技术，产品是产品，有好技术不一定能出好产品，但是没有好技术一定出不了好产品，当然，忽悠除外。令人眼前一亮的是，

Memblaze对产品这个概念还是非常有感觉的，深知“个性”对于一款产品来说是多么重要。

在我们的脑海里，PCIE闪存卡就是一锤子买卖，比如，卡上焊上了多少容量的Flash颗粒，它就是多大容量了，比如1TB容量，如果用户不需要这么大容量，比如只需要500GB容量，那么厂商就不得不再去定制一批只焊了500GB容量颗粒的板子。这个问题谁都清楚，但是Memblaze是第一个提出方案并且成功商用的公司。

“琴键”技术是他们对这个技术的命名，通过将Flash控制器与Flash颗粒之间的连接方式从完全PCB布线转为插槽的形式，然后通过生产不同容量的子卡（琴键）插在插槽上从而生成不同整体容量的闪存卡产品，如图3-43和图3-44所示。Memblaze组合容量和规格如图3-45、图3-46所示。

图3-43 Memblaze的琴键技术

图3-44 Memblaze的琴键技术

图3-45 Memblaze的琴键技术不同组合容量

图3-46 Memblaze闪存卡其他规格

说明： Memblaze是一家让人钦佩的初创公司，在此我也得知他们后续有很长远的规划，也非常期待能够早日见到其更有特色的产品。

3.12 小结：网中有网，网中之网

我们用图3-47来作为本章的结束。

图3-47 三台计算机组成的网——网中有网

第4章 七星北斗——大话/详解七种**RAID**

■ RAID 1

■ RAID 2

■ RAID 3

■ RAID 4

■ RAID 5

■ RAID 5E

■ RAID 5EE

■ RAID 6

第3章介绍了磁盘的内部原理、构造以及外部的接口系统。但一块磁盘的容量是有限的，速度也是有限的。对一些应用来说，可能需要上百吉字节（GB）甚至几太字节（TB）大小的分区来存放数据。目前的磁盘单块容量最多能到1TB，这对于现代应用程序来说远远不够。

那么必须要制造单盘容量更大的硬盘么？为解决这个问题，人们发明了RAID技术，即Redundant Array of Independent Disks技术，中文意思是由独立的磁盘组成的具有冗余特性的阵列。既然是阵列，那一定需要很多磁盘来组成；既然是具有冗余特性的，那一定可以允许某块磁盘损

坏之后，数据仍然可用。下面我们就来看一下RAID是怎样炼成的。

4.1 大话七种**RAID**武器

话说几千年前，有位双刀大侠，左右手各拿一把大刀。开始的时候，他总是单独使用每把刀，要么用左手刀，要么用右手刀，但是总被人打败，郁闷至极，于是苦心悟道，静心修炼。他逐渐摸索出一套刀法，自称“合一刀法”，即双刀并用。外人看不见他的第二把刀，只能看到他拿着一把刀。他把两把刀的威力，合二为一，成为一把大刀！

而这种双刀合一的刀法，又可以分为两条路子。一条是常规路子，即这把合二为一的大刀，其实每出一招只有原来一把刀的威力，但是后劲更足了。一把刀顶不起来的时候，可以第二把刀上阵。对于敌人来说，还是只看见一把大刀。另一条是野路子，野路子往往效果很好，每出一招总是具有两把刀的威力之和，而且也具有两把刀的后劲！他实现这个野路子的方法，便是界内早有的思想——分而治之！也就是说他把一把刀又分成了很多细小的元素，每次出招时把两把刀的元素组合起来，所以不但威力大了，后劲也足了！

不过大侠的这个刀法有个致命的弱点，就是双刀息息相关。一旦其中一把刀有所损坏，另一把刀相应的地方也跟着损坏。如果一把刀完全失去效力，那么另一把刀也跟着失效。

双刀大侠一直到临终也没有收一个徒弟，不是因为他武艺不精，而是因为他的合一刀法在当时被认为是野路子，歪门邪路的功夫，所以郁闷一生。临终时他用尽自己最后一点力气在纸上写下了4句诗后，抱憾而终！

刀于我手不为刀，

横分竖割成龙绦。
化作神龙游天际，
龙在我心任逍遥！

这就是后世流传的“合一刀谱”！俗称“龙谱”。

世上最高的刀法在心中，而不是手上！双刀大侠练就的是一门“浩瀚”绝学，一招一式都是铺天盖地，势不可挡！

几百年后，七星大侠在修炼了磁盘大挪移神功和龙谱之后的某一天，他突然两眼发愣：“我悟到了！”然后奋笔疾书，成就了“七星北斗阵”这个空前绝后的阵式！RAID 0阵式就是这个阵式的第一个阵式！下面来看看这个阵式的绝妙之处吧！

4.1.1 RAID 0阵式

首先，这位七星大侠一定是对磁盘大挪移神功有很高的造诣，因为他熟知每块磁盘上面的磁性区域的构造，包括磁道、磁头、扇区和柱面等，这些口诀心法已经烂熟于心。在他看来，盘片就像一个蜂窝，上面的每一个孔都是一个扇区，可以说他已经参透了磁盘。其次，七星大侠一定是对合一刀法的精髓有很深的领悟，即他能领会双刀大侠那4句诗的含义，特别是第二句给了他很大的启发！“横分竖割成龙绦”，暗示着双刀大侠把他的刀在心中分割成了横条带和竖条带，所以叫“横分竖割”。分割完毕之后，双刀大侠把这些分割后所谓的“绦”，即细条带，在心中组合起来形成一条虚拟的“龙”，然后用龙来当作武器，即“龙在我心任逍遥”。

这显然给了七星大侠很大的启发，何不把几块磁盘也给“横分竖割”，然后组成“龙”呢？对，就这么干！七星大侠卖血换来两块磁盘，

找了个破庙，在后面搭了个草堆，成天摆弄他那两块用血换来的磁盘。白天出去要饭，晚上回来潜心钻研！他首先决定把两块磁盘都分割成条带，形成“条”，可是该怎么分好呢？合一刀法的思想主要有两条路：一条是懒人做法，不想动脑子，即威力小、后劲足那种；另一种是需要动脑子算的，即威力足、后劲也足那种。

第一种怎么实现呢？七星大侠冥思苦想，却发现被误导了。因为第一种根本不需要做“条”。双刀大侠的诗只是描述了威力巨大的第二种路子。所以三下五除二，七星大侠写出了RAID 0阵式中的第一个套路：累加式。也就是说，磁盘还是那些磁盘，什么都不动，也不用“横分竖割”。数据来了，先往第一块磁盘上写。等写满之后，再往第二块上写。然后将这两块磁盘在心中组成一条龙，这就符合了合一刀法的思想。只不过这条龙威力比较弱，因为每次IO只用到了一块磁盘，另一块磁盘没有动作。但是这条龙的后劲，比单个磁盘足了，因为容量相对一块磁盘来说，增大了。

第一套路子实现了，可第二套路子就难了。其实磁盘已经被分割了，扇区不就是被分割的么？但是一个扇区只有512B大小，这不符合合一刀法。因为合一刀法中是“条”而不是“粒”，分割成粒的话，不仅开销太大，而且组合起来也很困难。所以七星大侠决定完全照搬合一刀法的思想，但是又不能丢弃磁盘已经分割好的扇区，所以七星绞尽脑汁想出这么一个办法，如图4-1和图4-2所示。

图4-1 正常的4块硬盘

图4-2 被分割的4块硬盘

图4-1所示的是4块普通硬盘，其上布满了扇区。扇区是实实在在存在于盘片上的，具有自己的格式。图4-2所示的是引入分割思想之后的

硬盘。由于许多文件系统或者卷管理软件都使用块而不是扇区作为基本存储单元，所以图4-2中也使用由4个扇区组成的块作为基本单元。不同磁盘的相同偏移处的块组合成Stripe，也就是条带。

块的编号也是以横向条带方向开始一条一条的向下。这样，对于一个全新的文件系统和RAID 0磁盘组，如果有大块数据写入时，则数据在很大几率上可以以条带为单位写入。也就是说数据被分成多块写入4个硬盘，而不是单硬盘系统中的顺序写入一个硬盘，这就大大提高了速度。图4-3所示的为多块磁盘组成的逻辑磁盘示意图。

图4-3 心中之龙

提示： 磁盘上实实在在存在的只有扇区结构，Stripe并不是一个实实在在的结构，它只是由程序根据算法公式现套现用的。就像戴了一副有格子的眼镜看一张白纸，那么会认为这张白纸被格式化了，其实并没有。另外，条带化之后的多块硬盘，数据是被并行写入所有磁盘的，也就是多管齐下，而不是横向写满一个条带，再写下一个条带。

七星大侠就这样埋头苦苦思考了整整1年，基于合一刀法的横分竖割的思想，完成了“七星北斗阵”的第一个阵式——RAID 0阵式。

4.1.2 RAID 1阵式

花开七朵，各表其一。话说七星在完成RAID 0阵式之后，并没有沾沾自喜，而总是想在合一刀法上有所创新。

提示： RAID 0阵式纵然威力无比，但弱点也很明显：一旦其中一块磁盘废掉，整个阵式将会被轻易攻破。因为每次出招靠

的就是“合一”，如果任意一块坏掉，也就没有“合”的意义了。也就是说，数据被我在心中分割，本来老老实实写到一块盘就完事了，可为了追求威力，非要并发写盘，第一、三、五、.....块数据写到了1号盘，第二、四、六、.....块数据写到了2号盘。但是对于外界来说，会认为是把数据都写到了我心中的一块虚拟盘上。这样不坏则已，一旦其中任何一块磁盘损坏，就会数据全毁，因为数据是被分割开存放在所有磁盘上的。不行，太不保险了。为了追求威力，冒的险太大，要想个稳妥的办法。

于是七星再次冥思苦想，终于创出了RAID 1阵式！

这话要说到800年前，有位“独行侠”，心独身独终日孤单一人。据称他每次出招从来不用双手，总是单手打出单掌，练就了一门“独孤影子掌”。虽说此掌法威力不高，但是自有其妙处。每当他敌不过他人，单掌被击溃的时候，就会立即换用另一只从来都没用过的掌继续出招。这一绝学往往令自以为已经占了上风的敌人在还没有回过神来的情况下，就被打个落花流水！不但他的掌法绝妙，就连他的身法都达到了炉火纯青的地步。他能修炼出一个影子，这个影子平时总是跟随着他，他做什么，影子就做什么。一旦真身损毁，其影子便代替他的真身来动作。这位独行侠遗留的“孤独影子掌”秘笈如下：

心朦胧，掌朦胧。

掌由独心生。

身朦胧，影朦胧。

身影心相同。

花朦胧，夜朦胧。

独饮赏月容。

灯朦胧，人朦胧。

此景何时休？

独行侠的这段诗句不难理解，孤独给了他灵感，身独心也独，如此练就的功夫，也是独孤残影。最后一句说出了大侠的无奈，其实他也不想孤独，但是没人能理解他。

七星大侠领悟了独行侠的苦衷，感受了他的心境。独行侠练就的是一门“无奈”绝学，处处体现着凄惨与潦迫。只有残了，才能重获新生。一只掌断了，另一只掌才能接替，这是何等凄惨？简直凄惨至极！不过往往孤独凄苦的人都很注意自保，虽然招式的威力是最小的，但是这门学问是武林中用于自保的最佳选择。

七星大侠没有理由不选择这门自保神功来解决他在RAID 0阵式中的破绽，也就是安全问题。毫无疑问，RAID 0是强大的，但是也是脆弱的，一点点挫折就足以让RAID 0解体。

七星大侠决定完全抛弃RAID 0的思想，采用独行侠的思想。将两块磁盘中的一块用于正常使用，另一块用作正常使用磁盘的影子。影子总是跟随主人，主人做什么，影子就做什么。工作盘写了一个数据，影子盘在相同位置也写上了数据。读数据的时候，因为数据有两份，可以在工作盘读，也可以到影子盘读，所以增加了并发性。即修炼这个阵式的人，可以同时应付两个敌人的挑衅，自身应付一个，影子应付一个，这无疑是很高明的！

但是应付一个敌人的时候，不像RAID 0阵式那样可以同时使用多块磁盘，只能使用一块磁盘。当其中一块磁盘坏掉，或者其中一块磁盘上某个区域坏掉，那么对应影子盘或者影子盘上对应的位置便会立即接替

工作盘，敌人看不出变化。可能独行侠一生都没有遇到同时和两个对手过招的情况，所以在他的秘笈中，并没有体现“并发读”这个功能，只体现了安全自保。

然而七星并没有全面抛弃双刀大侠的思想，而是保留了双刀的精华，即“横分竖割”的基本思想，抛弃了他的算法，即鲁莽而不计后果的并发往各个磁盘上写数据的方法。所谓算法，也即指大侠对付敌人招数的时候，在心中盘算的过程。心算的速度远远快于出招的速度，所以心算引发的延时并不会影响出招。现在江湖人士也大多都是精于钻研算法，而只有制造兵器的铁匠才去钻研如何用料，考虑如何才能减轻兵器重量而不影响兵器的硬度和耐磨度等。

可以说，兵器的材质、设计加上大侠们精心的算法才形成了江湖上形形色色的功夫秘笈！而材质在很大程度上发展是很慢的，想有突破非常困难。但是算法就不同了，大侠可以研究出各种使用兵器的方法，将兵器用得神乎其神！磁盘的转速、磁密度、电路等，虽然一直在提升，但是终究太慢。所以出现了以七星大侠为代表的算法派，他们苦研算法，用来提高磁盘的整体性能。假想某天一旦某个铁匠造出了屠龙刀、倚天剑这般的神器，我想七星这等大侠也就无用武之处了。可惜这两把神器已经在自相残杀中玉石俱焚了。

七星大侠最后给这个阵式起名叫做“RAID 1”。图4-4显示了RAID 1组成的逻辑磁盘。

图4-4 RAID 1示意图

但是七星也深深认识到，这个RAID 1阵式还是有两个大弱点。

在修炼的时候，速度会稍慢，因为每次修炼，除了练真身之外，还

要练影子。不然影子不会，出招的时候影子就无法使用。这会对实际使用有一定影响，数据写到工作盘上，也必须写到影子盘上一份。

虽然自己有个影子，但是影子没有给真身增加后劲。真身累了，影子也会被拖累。不管修炼了几个影子，整体的耐力和体力只等于其中一个的体力耐力，也就是真身的体力和耐力。整体的体力和耐力都被限制在体力耐力最小的影子盘或者真身上。也就是说RAID 1提供的最大容量等于所有组成RAID 1的磁盘中容量最小的一块，剩余容量不被使用，RAID 1磁盘组的写性能等于所有磁盘中性能最低的那块磁盘的写性能。

七星看了看RAID 0，又看了看RAID 1，一个鲁莽急躁但威力无比，一个独孤残苦自嘲自保。唉！呜呼哉！！七星心想，我怎么走了两个极端呢？不妥，不妥，二者皆不合我意！于是，七星大侠又开始了苦心钻研，这一下就是2年！

4.1.3 RAID 2阵式

史话： 话说明末清初时期，社会动荡，英雄辈出。有这么一位英雄，号称“优雅剑侠”，他持双手剑，得益于流传甚广的合一刀法，并加入了自己的招式，修炼成了一套“合一优雅剑法”。剑侠深知合一刀法的鲁莽招式，虽然威力巨大，但必会酿成大祸。所以他潜心研究，终于找到一种办法，可以避免合一的鲁莽造成的不可挽回的祸患。他分析过，合一之所以鲁莽就是因为他没有备份措施，兵器有任何一点损坏都会一损俱损。那么是不是可以找一种方法，对兵器上的每个条带都做一个备份，就像当年独行侠那样？但是又不能一个对一个，那就和独行侠无异了。

剑侠的脑子很好用，他从小精于算术，有常人不及之算术功夫。如今他终于发挥出他的算术技能了，他先找来一张纸，然后把他的两把剑和这张纸，并排摆在地上，然后对剑和纸进行横分竖割，然后一一对照，将第一把剑的第一格写上一个1，然后在第二把剑的相同位置上写上一个0，然后在纸的对应位置上算出前二者的和，即 $1+0=1$ 。然后剑侠设想，一旦第一把剑被损坏，现在只剩第二把剑和那张记满数字的纸，剑侠恍然大悟：原来如此精妙！

为什么呢？虽然第一把剑损坏，但是此时仍然可以出招，因为第一把剑上的数字可以用纸上对应位置的数字，减去第二把剑对应位置上的数据！也就是 $1-0=1$ ，就可以得出第一把剑上已经丢失的数字！而在敌人看来，仍旧是手持一把大剑，只不过威力变小了，因为每次出招都要计算一次。而且修炼的时候也更加难了，因为每练一招就要在纸上记录下双剑之和，而且还需要用脑子算，速度比合一刀法慢了不少。

哇哈哈，剑侠仰天长笑！他给自己的剑法取名“优雅合一剑法”，意即他的剑法比合一刀法虽然威力不及，但也差不多。最重要的是，他克服了合一刀法鲁莽不计后果的弱点，所以要比合一刀法来得优雅。但是这个剑法也有弱点，就是他额外增加了一张纸和用了更多的脑筋来计算。脑子计算倒是不成问题，努力学习算法便可，但是额外增加了一张纸，这个难免有些遗憾，但是也没有办法，总比独行侠那一套自保好得多。自保的代价是修炼一个平时几乎用不到的影子，是一比一。优雅合一剑法是二比一，降低了修炼的代价，而威力却较合一刀法没减多少。

然而，这套剑法虽然声名大噪，但是优雅剑侠还是被一个突如其来的问题一直折磨着，直到临终也没有想出办法解决。

疑问：如果我使用3把剑、4把剑、5把剑，这套剑法还奏效

么？因为3把剑的数字之和，就不是一个数字，而是两个数字了，比如 $1+0+1=10$ 。而这套剑法只有一张纸，一个格不能放两个数字，这样就必须再加一张纸，这样不就和独行侠那一套无异了么？比例太高，不妥。所以优雅剑侠一直再考虑这个问题，临终前留下一段诗句，也抱憾而终。

独行合一皆非道，
二者中庸方优雅。
加减算术勤思考，

世间正道为算法！

优雅剑侠这段诗的最后一句，指明了后人若要解决这个问题，必须要找到一种算法。不管多少个数字，如果掩盖一个数字，可以将其他数字代入这个算法，就可以得到被掩盖的数字的值。这在当时简直就是不可能的事。“世间正道为算法”这句话后来被作为推动武林发展的一句至理名言。随着近代西方科技传入中国，这种算法终于被人了解了！他是如此简单而美丽！他改变了整个世界！

峰回路转，七星大侠在优雅剑法的基础上，把剑换成了磁盘。那张记录数字之和的纸，七星也改用磁盘来记录。这样，组成了一个三磁盘系统，两块数据盘，一块所谓“校验盘”。当数据损坏时，根据校验盘上的数字，可恢复损坏磁盘上的数字。两个磁盘系统每次只能传输2路数据，因为数据盘就两块，而每块磁盘每次就传输出去一路。

RAID 2的具体实现如图4-5所示。

图4-5 RAID 2的具体实现

看看RAID 2的具体实现。值得研究的是，七星大侠并没有使用加减法来进行校验，而是用了一种算法复杂的所谓“汉明码”来校验。这可不是信手拈来，而是有一定原因的。用加减算法进行校验，并没有对数据纠错的能力。加减法情况下，比如 $1+0+1=10$ ，这段数据在从磁盘被传输给控制器的时候，校验位会一同传输，即数据位101，校验位10，此时经过控制器的校验，他会算出 $1+0+1=10$ ，和一同传过来的校验位进行比对。如果相同，则证明数据都无误。但是此时如果在传输的过程中，电路受到干扰，数据位其中有一位畸变了，比如变成111了，也就是中间那个0变成1了，其他不变，此时控制器进行计算 $1+1+1=11$ ，和一同传过来的10不同，那么控制器会怎么认为？它可以认为数据位全部正确，而校验位被畸变，也可以认为数据位被畸变，校验位正确。所以根本不能判断到底是哪种情况，所以不能修复错误。而七星引入了汉明码，汉明码的设计使得接收方可以判断到底是哪一位出错了，并且能修正一位错误，但是如果两位都错了，那么就不能修正了。

Hamming Code ECC（汉明码错误检测与修正）

RAID 2算法的复杂性在于它使用了很早期的纠错技术——汉明码（Hamming Code）校验技术。现在就来看一下汉明码的算法。

汉明码在原有数据位中插入一定数量的校验位来进行错误检测和纠错。比如，对于一组4位数据编码为例，汉明码会在这4位中加入3个校验位，从而使得实际传输的数据位达到7位，它们的位置如图4-5所示。

需要被插入的汉明码的位数与数据位的数量之间的关系为 $2^P \geq P + D + 1$ ，其中P代表汉明码的个数，D代表数据位的个数。比如4位数据，加上1就是5，而能大于5的2的幂数就是3（ $2^3=8$ ， $2^2=4$ ）。所以，7位数据时需要4位汉明码（ $2^4 > 4 + 7 + 1$ ），64位数据时就要7位汉明码

($27 > 64 + 7 + 1$)。

在RAID 2中，每个IO下发的数据被以位为单位平均打散在所有数据盘上。如图4.5中所示，左边的为数据盘阵列，如果某时刻有一个4KB的IO下发给这个RAID 2系统，则这4KB中的第1、5、9、13等位将被存放在第一块数据盘的一个扇区中，第2、6、10、14等位被存放在第二个磁盘的对应条带上的扇区，依此类推。这样，数据强行打散在所有磁盘上，迫使每次IO都要全组联动来存取，所以此时要求各个磁盘主轴同步，才能达到最佳效果。因为如果某时刻只读出了一个IO的某些扇区，另一些扇区还没有读出，那么先读出来的数据都要等待，这就造成了瓶颈。主轴同步之后，每块磁盘盘片旋转同步，某一时刻每块磁盘都旋转到同一个扇区偏移的上方。同理，右边的阵列（我们称之为校验阵列）则存储相应的汉明码。

RAID 2在写入数据块的同时还要计算出它们的汉明码并写入校验阵列，读取时也要对数据即时地进行校验，最后再发向系统。通过上文的介绍，我们知道汉明码只能纠正一个位的错误，所以RAID 2也只能允许一个硬盘出问题，如果两个或以上的硬盘出问题，RAID 2的数据就将受到破坏。

RAID 2是早期为了能进行即时的数据校验而研制的一种技术（这在当时的RAID 0、1等级中是无法做到的），从它的设计上看也是主要为了即时校验以保证数据安全，针对了当时对数据即时安全性非常敏感的领域，如服务器、金融服务等。但由于校验盘数量太多、开销太大及成本昂贵，目前已基本不再使用，转而以更高级的即时检验RAID所代替，如RAID 3、5等。

七星大侠现在已经创造了三种阵式了，根据合一刀法所创的RAID

0，根据独孤掌所创的RAID 1，根据优雅合一剑法所创的RAID 2。而七星大侠的郁闷之处和当年优雅剑侠一样，就是苦于找不到一种一劳永逸的绝妙算法，一种集各种优点为一身而且开销小的算法。

4.1.4 RAID 3阵式

话说到了清末，清政府开展洋务运动，师夷长技以治夷。还别说，真引入了不少好技术，比如布尔逻辑运算式。这话要从布尔说起，布尔有一次在家捣鼓继电器，他将多个继电器时而串联，时而并联，时而串并一同使用，逐渐摸索出一些规律。比如两个继电器在串联时，必须同时闭合两个开关，电路才能接通，灯泡才能亮。如果把开关闭合当作1，开关关闭当作0，灯泡点亮当作1，灯泡不亮当作0，那么这种串联电路的逻辑就可以这样写： $1 \text{ 和 } 1 = 1$ 。也就是两个开关都闭合，灯泡才能亮（等于1）。

然后，他还发现一个逻辑，如果在这个串联电路上增加一个元件，如果两个开关都闭合的时候，电路反而是断开的。有人说不可能，那么请仔细想一想，闭合开关电路断开，这有什么难的么？完全可以通过继电器来实现，比如电路闭合之后，电磁铁通电，把铁片吸引下来，而这个铁片是另一个电路的开关，铁片下来了，另一个电路也就断开了，所以通过把这两个电路组合，完全可以得到这种逻辑： $1 \text{ 和 } 1 = 0$ 。

还有一种逻辑，就是当两个开关任意一个闭合时，电路就通路，也就是并联电路。这种逻辑可以这么表达： $1 \text{ 或 } 1 = 1$ ， $1 \text{ 或 } 0 = 1$ 。经过多种组合，布尔得到了 $1 \text{ 或 } 1 = 1$ 、 $1 \text{ 或 } 0 = 1$ 、 $1 \text{ 和 } 1 = 1$ 、 $1 \text{ 和 } 0 = 0$ 。

这就是4种基本逻辑电路。这种“和”、“或”的运算，很多人都不理解。人们理解的只是加减运算，因为加减很常用。人们不理解的原因就

是不知道除了加减算术之外，还有一种叫做“逻辑”的东西，也就是因果的运算。人们往往把1当成数量，代表1个，而在因果率中，1不代表数量，它只代表真假，其实我们完全可以不用1这个符号来代表真，我们就用中文“真”代表真，可否？

当然可以，但是因为笔画太多，不方便，还是用1和0代表真假比较方便。其实磁盘上的数据，也不是1就代表1个，而是1代表磁性的取向，因为磁性只有两个取向，仿佛对称就是组成宇宙的基石一样，比如正、负，对、错等。当因果率被用数学方程式表达出来并赋予电路的物理意义之后，整个世界也就进入了新世纪的黎明，这个世纪是计算机的世纪。

提示：从数学到物理意义，我们仿佛看出点什么来，现代量子力学那一大堆数学式，折服了太多的科学家，包括爱因斯坦，到他去世前，爱因斯坦都没有理解量子力学所推演出来的数学式子在物理上到底代表了什么意义。而且直到21世纪，也没有人给予这些式子以“目前”可理解的物理意义。我们可以想象一下布尔逻辑算式公布的时候，物理意义到底是什么？没人知道，甚至布尔自己估计也不知道，就只是一对式子而已。直到有一天一个人在家捣鼓继电器，突然风马牛不相及地想到了，这不就是布尔逻辑么？从此，数字电路，计算机时代，改变了我们的世界。

七星在学习了布尔逻辑算式之后，也是稀里糊涂地把它用在RAID 2那一直困扰他的问题上面，看看能否有所突破。

布尔运算中有一个XOR运算，即 $1 \text{ XOR } 0 = 1$ ， $1 \text{ XOR } 1 = 0$ ， $0 \text{ XOR } 0 = 0$ 。布尔也总结出了与加法结合率、加法交换率等类似的逻辑运算

率，并发现了一些规律：

$$1 \text{ XOR } 0 \text{ XOR } 1 = 0$$

$$0 \text{ XOR } 1 \text{ XOR } 0 = 1$$

假如第一个式子中，中间的0被掩盖，完全可以从结果推出这个被掩盖的逻辑数字。不管多少位，进行逻辑运算之后还是一位。仔细一想也是理所当然的，逻辑结果只有两个值，不是真，就是假，当然只用一位就可以代替了。

提示： 大家可以自己算算，不管等式左边有多少位进行运算，这个规律都适用。但是在加减法中，若要保持等式右边有一位，则左边参与运算的只能是 $1+0$ 或者 $0+1$ ，再多一个数的话，右边就是两位了。但是逻辑运算中等式右边永远都是一位！就是如此绝妙。为什么如此精妙呢？没人能解释为什么，就像问为什么有正电荷、负电荷一样，它们到底是什么东西，谁也说不清。

七星大侠开始并不觉得这是真的，他反复演算，想举出一个伪证，可是徒劳无功。七星不得不为布尔的绝学所折服！同时也为西方发达的基础科学所赞叹！

至此，困扰七星大侠多年的关于算法的问题，终于随着西方科学的介入，得以顺利解决！而且解决得是那么完美，那么畅快！

七星立即决定投入其下一个阵式RAID 3的创立过程中。他发狂似地抛弃了那冗余的让人看着就不顺眼的RAID 2的几块校验盘，只留下一块。按照布尔的思想，数据盘的每一个位之间做XOR运算，然后将结果写入校验盘的对应位置。这样，任何一块数据盘损坏，或者其中的任何

一个扇区损坏，都可以通过剩余的位和校验位一同进行XOR运算，而运算的结果就是这个丢失的位。8位一起校验可以找出一个丢失的字节，512字节一起校验就可以找到一个丢失的扇区。

做到这里，已经算是成功了，但是七星还不太满足，因为他还有一桩心事，就是RAID 2中数据被打得太散了。七星大侠索性把RAID 3的条带长度设置成为4KB，这样刚好适配了上层的数据组织，一般文件系统常用的是以4KB为一个块。如果用4块数据盘，则条带深度为两个扇区或者1KB。如果用8个数据盘，则条带深度为1个扇区或者说512B。总之，要保持条带长度为上层块的大小。上层的IO一般都会以块为单位，这样就可以保证在连续写的情况下，可以以条带为单位写入，大大提高磁盘并行度。

七星在RAID 3阵式中，仍旧保持RAID 2的思想，也就是对一个IO尽量做到能够分割成小块，让每个磁盘都得到存放这些小块的机会。这样多磁盘同时工作，性能高。所以七星在RAID 3中把一个条带做成4KB这个魔术值，这样每次IO就会牵动所有磁盘并行读写。到此我们了解了，RAID 2和RAID 3都是每次只能做一次IO（在IO块大于Block SIZE的时候），不适合于要求多IO并发的情况，因为会造成IO等待。RAID 3的并发只是一次IO的多磁盘并发存取，而不是指多个IO的并发。所以和RAID 2一样，适合IO块大、IO SIZE/IO PS比值大的情况。

提示： 在极端优化的条件下，RAID 3也是可以做到IO并发的。控制器向一块磁盘发送的读写指令，其中包含一个所要读取扇区的长度，如果下一次IO与本次IO在物理上是连续的（连续IO），此时如果控制器做了极端的优化，则可以将这两次IO合并起来，向磁盘发送的每个IO指令中包含了两次上层IO的数据，这样也算是一种并发IO。当然，这种优化不仅仅可以在磁

盘控制器这一层实现，其实文件系统层也可以实现。

提示： RAID 3和RAID 2一样，要达到RAID 3的最佳性能，需要所有磁盘的主轴同步。也就是说，对于一块数据，所有磁盘最好同时旋转到这个数据所在的位置，然后所有磁盘同步读出来。不然，一旦有磁盘和其他磁盘不同步，就会造成等待，所以只有主轴同步才能发挥最大性能。

总结一下，RAID 3相比RAID 2校验效率提升，成本减少（使用磁盘更少了）。缺点是不支持错误纠正了，因为XOR算法无法纠正错误。但是这个缺点已经不重要，发生错误的机会少之又少，可以完全靠上层来处理错误了。正可谓：

与非异或同，
一语解千愁。
今朝有酒醉，
看我数风流！

下面说明关于RAID 3的校验盘有没有瓶颈的问题。

不妨用一个例子来深入理解一下RAID 3。通过刚才的讲解，大家知道了RAID 3每次IO都会分散到所有盘。因为RAID 3把一个逻辑块又分割成了N份，也就是说如果一个逻辑块是4KB（一般文件系统都使用这个值），在有5块盘的RAID 3系统中其中4块是数据盘，1块是校验盘。这样，把4KB分成4块，每块1KB，每个数据盘上各占1块，也就是两个扇区。而文件系统下发的一个IO，至少是以一个逻辑块为单位的，也就是不能IO半个逻辑块的单位，也就不可能存在一个IO，大小是小于4KB的，要么是1个4KB，要么是N个4KB。但这只是针对文件系统下发的IO，磁盘控制器驱动向磁盘下发的IO最细粒度可以为一个扇区。这样，

就保证了文件系统下发的一次IO，不管多大都被跨越了所有数据盘。

读又分成连续读和随机读。连续读指的是每个IO所需要提取的数据块在序号上是连续的，磁头不必耗费太多时间来回寻道，所以这种情况下寻道消耗的时间就很短。我们知道，一个IO所用的时间约等于寻道时间加上旋转延迟时间再加上数据传输时间。 $IOPS = 1 / (\text{寻道时间} + \text{旋转延迟时间} + \text{数据传输时间})$ ，由于寻道时间相对于传输时间要大几个数量级，所以影响IOPS的关键因素就是寻道时间。而在连续IO的情况下，仅在换磁道时需要寻道，而磁道都是相邻的，所以寻道时间也足够短。在这个前提下，传输时间这个分母就显示出作用来了。由于RAID 3是一个IO，必定平均分摊到了N个数据盘上，所以数据传输时间是单盘的1/N，从而在连续IO的情况下，大大增加了IOPS。而磁盘总体传输速率约等于IOPS乘以IO SIZE。不管IO SIZE多大，RAID 3的持续读性能几乎就是单盘的N倍，非常强大。

再看看持续写，同样的道理，写IO也必定分摊到所有数据盘，那么寻道时间也足够短（因为是持续IO）。所以写的时候所耗费的时间也是单盘的1/N，因此速率也是单盘的N倍。有人说RAID 3的校验盘是热点盘，是瓶颈。理由是RAID 3写校验的时候，需要像RAID 5一样，先读出原来的校验块，再读出原来的数据块，接着计算出新校验，然后写入新数据和新校验。实际上RAID 3中每个IO必定要改动所有数据盘的数据分块。因为一个文件系统IO的块已经被分割到所有盘了，只要这个IO是写的动作，那么物理磁盘上的所有分块，就必定要全部都被更新重写。既然这样，还有“旧数据”和“旧校验”的概念么？没有了，因为这个IO上的所有分割块需要全部被更新，包括校验块。数据在一次写入之前，控制器就会计算好校验块，然后同时将数据块和校验块写入磁盘。这就没有了什么瓶颈和热点的区别！

RAID 4是有热点盘，因为RAID 4系统处理文件系统IO不是每次都会更新所有盘的，所以它必须用RAID 5的那个计算新校验的公式，也就是多出4个操作那个步骤，所以当然有瓶颈了！要说RAID 3有热点盘，也行，所有盘都是热点盘，数据、校验，所有盘，对RAID 3来说，每次IO必将牵动所有盘，那么就可以说RAID 3全部都是热点盘！

再来看看RAID 3的随机读写。所谓随机IO，即每次IO的数据块是分布在磁盘的各个位置，这些位置是不连续的，或者连续几率很小。这样，磁头就必须不断地换道，换道操作是磁盘操作中最慢的环节。根据公式 $IOPS = 1 / (\text{换道时间} + \text{数据传输时间})$ ，随机IO的时候换道时间很大，大出传输时间几个数量级，所以传输得再快，翻10倍也才增高了一个数量级，远不及换道时间的影响大，所以此时可以忽略传输时间的增加效应。由于一次IO同样是被分割到了所有数据盘，那么多块盘同时换道，然后同时传输各自的那个分块，换道时间就约等于单盘。其传输时间是单盘的 $1/N$ ，而传输时间带来的增效可以忽略。所以对于随机读写的性能，RAID 3并没有提升，和单盘一样，甚至不及单盘。因为有时候磁盘不是严格主轴同步的，这样换道慢的磁盘会拖累其他磁盘。

再来看看并发IO。显然，RAID 3执行一次IO必将牵动占用所有盘，那么此时其他排队的IO就必须等待，所以RAID 3根本就不能并发IO。

注意： 上文中的“IO”均指文件系统下发的IO，而不是指最终的磁盘IO。

4.1.5 RAID 4阵式

七星自从学习了西方先进的基础科学之后，一发而不可收。以前已

经是以钻研为乐，现在成了以钻研为生。以前饿了还知道去要饭吃，现在七星已经感觉不到饿了，只要有东西让他钻研，就等于吃饭了。

话说某天七星正在闭目思考修炼，他回想起了双刀时代的辉煌、独行时代的凄苦和优雅剑时代的中庸之乐。往事历历在目，再看看如今已经是穷困潦倒的自己，他不禁潸然泪下，老泪纵横。

他给上面的三种思想，分别划分了门派，RAID 0属于激进派，RAID 1属于保守派，RAID 2，RAID 3属于中庸派。中庸派的思想一方面汲取了激进派的横分竖割提高威力的做法，一方面适当地降低威力来向保守派汲取了自保的经验，而创立了引以为豪的校验盘的绝妙技术。

七星想着激进派似乎已经没有什么可以让中庸派值得借鉴的地方了，倒是保守派的一个关键技术中庸派还没有移植过来，那就是同时应付多个敌人的技术。虽然当年独行侠根本就没有意识到他的独孤影子掌可以同时应付两个敌人，因为独行侠一生都没有同时和两个人交过手。虽然独孤掌的秘笈中也没有提及这门绝招，但是七星凭他积累多年的知识和经验，强烈地感觉到并发IO早在独孤掌时代就已经被实现了，只不过没有被记载，而且一直被人忽略！要想有所突破就必须突破这一关！想到这里，七星立即再次开始了他的实验。

RAID 2阵式中，数据块被以位为单位打散在多块磁盘上存储，这种设计确实应该被淘汰了，且不说IO设计合理与否，看它校验盘的数量就让人气不打一处来。那么再看看RAID 3，在RAID 3的IO设计中还是走了RAID 2的老路子，也就是一次IO尽量让每块磁盘都参与，而控制器的一次IO数据块不会很大，那么想让每块磁盘都参与这个IO，就只能人为地减小条带深度的大小。

事实证明这种IO设计在IOSIZE/IOPS（比值）很大的时候，确实效果明显。但在现实应用中，很多应用的IOSIZE/IOPS都很小，比如随机小块读写等，这种应用每秒产生的IO数目很大，但是每个IO所请求的数据长度却很短。如果所有磁盘同一时刻都被一个IO占用着，且不能并发IO，只能一个IO一个IO的来做，由于IO块长度小，此时全盘联动来传输这个IO，得不偿失，还不如让这个IO的数据直接写入一块磁盘，空余的磁盘就可以做其他IO了。

要实现并发IO，就需要保证有空闲的磁盘未被IO占用，以便其他IO去占有磁盘进行访问。唯一可以实现这个方法就是增大条带深度，控制器的一个IO过来，如果这个IO块小于条带深度，那么这次IO就被完全“禁锢”在一块磁盘上，直接就写入了一个磁盘上的Segment中，这个过程只用到了一块磁盘。而其他IO也可以和这个IO同时进行，前提是其他IO的目标不是这个IO要写入或读取的磁盘。所以实现IO并发还需要增大数据的随机分布性，而不要连续在一个磁盘上分布。这里七星大侠忽略了一个非常重要的地方，下面会看到。

在这些分析的基础上，七星将RAID 3进行了简单的改造，增大了条带深度，于是便创立了一个新的阵式，名曰RAID 4。

四海一家

先	并
来	肩
后	携
到	手

先	并
进	存
先	并
出	取
天	海
经	誓
地	山
义	盟

4.1.6 RAID 5阵式

话说七星大侠正在为创立了RAID 4阵式而欢喜的时候，麻烦来了。很多江湖上的朋友都给他捎信说，修炼了RAID 4阵式之后，在IO写的时候，好像性能相对于RAID 3并没有什么提升，不管IOSIZE/IOPS的值是多少。这个奇怪的问题，让七星大侠天天如坐针毡、茶饭不思，终日思考这个问题的原因。他不停地拿着两块磁盘和一张纸（校验盘）比划。时间一长，七星有一天突然发现，纸已经被他画的不成样子了，需要另换一张。这引起了思维活跃的七星大侠的思考，“并发IO，并发IO，并发IO，.....”，他不停地嘴里念叨着。突然他两眼一睁，骂了一句之后，开始奋笔疾书。

七星大侠想到了什么让他恍然大悟呢？原来，七星经过思考之后，发现RAID 4确实是他的一大败笔，相对RAID 3没有什么性能提升，反

而误人子弟，浪费了很多人的时间去修炼一个无用的功夫。

七星创立RAID 4时，太过大意了，竟然忽略了一件事情。每个IO写操作必须占用校验盘，校验盘每一时刻总是被一个IO占用，因为写数据盘的时候，同时也要读写校验盘上的校验码。所以每个写IO不管占用了哪块数据盘，校验盘它是必须占用的，这样校验盘就成为了瓶颈，而且每个写入IO都会拖累校验盘，使得校验盘没有休息的时间，成了“热点盘”，非常容易损坏。

没有引入校验功能，数据盘可以被写IO并发，引入了校验功能之后，数据盘还可以并发，但是校验盘不可以并发，所以整体上还是不能并发。除非不使用校验盘，不过那就和RAID 0没什么区别了。所以七星在RAID 4上掉进了一个误区，如今他终于醒悟了。

创立的RAID 4什么都不是，不伦不类。七星这个郁闷啊！为了实现真正的写IO并发，他这次是豁出去了，一定要创立新的阵式！

思考： RAID 4的关键错误在于忽略了校验盘。每个IO不管目标在哪个数据盘，但是一定要读写校验盘。而校验盘只有一块，不读也得读！那如果有两块校验盘，它能否随机选择一块来读写？不行，这两块校验盘之间也要同步起来，类似RAID 1，这样开销太大，成本太高。

不妨作一下演绎，首先，我们的目标是并发IO。要并发IO，校验盘某一时刻必须可以被多个IO占用，这是必须的，否则就不是并发IO。但是“校验盘某一时刻可以被多个IO占用”，这不简直是扯淡么？一块磁盘怎么可以同时被多个IO占用呢？所以七星下了结论，中庸派不可能实现并发IO。

结论下了，七星也病了，彻彻底底的病倒了。他不甘心，在他心中一定有一个完美的阵式。他拿着那张已经快被画烂的纸，气愤至极，将纸撕成了两半。碎片就像七星那破碎的心，飘飘落下，不偏不倚，正好分别落到了地上的两把剑上，分别盖住了剑的一半。七星看着这情景，一发愣，仿佛冥冥中一直有个神仙在指引着他似的，又一次让七星茅塞顿开，恍然大悟：“老天助我啊！哇哈哈哈哈哈哈！”

七星疯了一般从炕上滚落下来，他又找来一把剑，把纸撕成三块，分别盖住每把剑的三分之一。同样，4把剑，把纸撕成4块，盖住剑的四分之一。良久之后，七星仰天长叹：“完美，太完美了！”

七星赶紧静下心来，他深知，必须经过深思熟虑，才不会出现问題，不能重蹈RAID 4的覆辙。他花了半个月的时间，用树枝在地上画图演算，并仔细分析。一块磁盘同一时刻不能被多个IO占用，这是绝对真理，不可质疑的真理。那么以前也曾经想过，把校验盘做成多块，可否？也不好，不完美。这次把校验盘分割开，组合于数据盘之中，依附于数据盘，这样就完美地避开了那个真理。既然多个IO可以同时访问多块数据盘，而校验盘又被打散在各个数据盘上，那么就意味着多IO可以同时访问校验盘（的“残体”）。这样就大大增加了多IO并发的几率，纵使发生多个IO所要用的校验盘的“残体”可能在同一块数据盘上，这样还是可以IO排队等待的。

如果数据盘足够多，校验盘打散的部分就会分布得足够广泛，多IO并发的几率就会显著增大！他根据这个推断做实验，首先是两个数据盘。把纸撕成两半，分别盖住两把剑的一半，这样实际的数据盘容量其实是一把剑的容量，校验盘容量也是一把剑的容量，它们分别占了总容量的二分之一。由于2块盘的RAID 5系统，对于写操作来说不能并发IO，因为一个IO访问其中一块盘的数据的时候，校验信息必定在另一块

盘，必定也要同时访问另一块盘。同样，3块盘的RAID 5系统也不能并发IO，最低可以并发IO的RAID 5系统需要4块盘，而此时最多可以并发两个IO，可以算出并发几率为0.0322。更多磁盘数量的RAID 5系统的并发几率将更高。图4-6为一个RAID 5系统的示意图。

图4-6 RAID 5系统示意图

七星这次可谓是红星高照，脸色红润，病态全无。他把这个新创立的阵式叫做RAID 5。正可谓：

心似剑，剑如心，
剑心合，方不侵。
分久必合合久分，
分分合合天地真！

RAID 5也不是那么完美无缺的，可以说RAID 5是继RAID 0和RAID 1之后，又一个能实现并发IO的阵式，但是比RAID 1更加划算，比RAID 0更加安全。RAID 5浪费的资源，在2块盘的系统中与RAID 1是一样的，都是二分之一。但是随着磁盘数量的增加，RAID 5浪费的容量比例越来越小，为N分之一，而RAID 1则永远是二分之一。

RAID 5和RAID 0都是利用条带来提升性能，但是它又克服了RAID 0的鲁莽急躁，对数据用校验的方式进行保护。但是RAID 5的设计思想，注定了它的连续读性能不如RAID 3，RAID 3由于条带深度很小，每次IO总是能牵动所有磁盘为它服务，对于大块连续数据的读写速度很快。但是RAID 5的条带深度比较大，每次IO一般只使用一块数据盘，而且通用RAID 5系统一般被设计为数据块都是先放满一个Segment，再去下一个磁盘的Segment存放，块编号是横向进行的。

RAID 5在随机读方面，确实是首屈一指的，这要归功于它的多IO并发的实现，这里指的是随机IO。也就是说RAID 3在IOSIZE值大的时候具有高性能，RAID 5在随机IOPS大的时候具有高性能。

RAID 5的一大缺点就是写性能较差。写性能差是中庸派的通病，其根本原因在于它们每写一扇区的数据就要产生其校验扇区，一并写入校验盘。尤其是更改数据的时候，这种效应的影响尤其严重。

RAID 5写的基本过程是这样的，新数据过来之后，控制器立即读取待更新扇区的原数据，同时也要读取这个条带上的校验数据。三者按照公式运算，便可得出新数据的校验数据，然后将新数据和新数据的校验数据写到磁盘。公式如下：

$$\text{新数据的校验数据} = (\text{老数据} \oplus \text{新数据}) \oplus \text{老校验数据}$$

鱼和熊掌不可兼得！最后七星总结了这么一句话。也就是说随机并发IO和写性能二者取其一。但是有些文件系统巧妙地减少了这种写惩罚，使得RAID 4的缺点被成功削减，从而将其优点显现了出来，本章后面会提到这种优化操作。

思考： RAID 5一次写的动作，其实要浪费掉3个其他动作，也就是要先读出老数据，读出老校验数据，然后写新数据和校验数据，这样只有“写新数据”是要完成的目的，而捎带了三个额外操作。纵观RAID 0和RAID 1此二者，RAID 0鲁莽，写就是写，不带任何考虑，所以速度最快。RAID 1自保，但是每次也只要写两次即可，只是额外多了一个操作。所以RAID 5和RAID 4在处理写方面是失败的。就连RAID 2和RAID 3都比RAID 5写性能强，因为它们的条带深度很小，任何一次正常点

的IO几乎会覆写所有盘，均会将这整个条带上的位都改变。所以RAID 2和RAID 3不用顾忌条带上是否还有未被更新的数据，所以它不会管老数据如何，只管从新数据计算出新校验数据，然后同时将数据位和校验位分别写到数据盘和校验盘，这样只用了两个操作，比RAID 5少了两次读的过程。

RAID 5E和RAID 5EE阵式

七星大侠推出RAID 5之后，得到了极为广泛的应用，江湖上的武林人士都在修炼，有些练成的大侠还各自创办了数据库、网站等生意，得益于RAID 5的随机IO并发特性，这些人赚了一大笔，生意火得一塌糊涂。然而，七星还是那个要饭的七星，剑还是那把剑，依然终日以钻研为乐，以钻研为生。

话说有一天，有个侠客专门找到了七星大侠，侠客请他到“纵横斋”煮酒畅饮。酒过三巡，菜过五味，侠客进入了正题，向七星叙述了一件事情。他说他已经炼成了RAID 5阵式，但是在使用的时候总是心里没底。其原因就是一旦一块磁盘损坏，虽然此时不影响使用，但是总有顾虑，不敢全力出招，就怕此时再坏一块磁盘，整个阵式就崩溃了。

他请求七星解决这个问题，临走的时候留下了几块市面上品质最好的硬盘和一些银子供七星研究使用。七星很是感动，几十年来从来没有一位江湖人士和他交流切磋过，也从来没有一个人来帮助过他。此景让他泪流满面，感动得不知说什么好。他向那位侠客道：“能交您这位豪杰人士，我七星此生无憾！”随后，七星又开始终日研究。这位大侠就是几十年后谱写降龙传说的张真人。

思考： 嗯，一旦一块磁盘损坏，此时这块盘上的数据已经不

复存在，但是如果有IO请求这块坏盘上的数据，那么可以用还存在的其他数据，校验出这块损坏的数据，传送出去。也就是说，损坏的数据是边校验边传送，现生成现传送的。而对于要写入这块盘的IO，控制器会经过计算，将其“重定向”到其他盘。所谓重定向并不是完全透明地写入其他盘，而是运用XOR进行逆运算，将写入的数据代入算式进行逆运算，得到的结果写入现存的磁盘上。

此时如果再有一块磁盘损坏，无疑阵式就要崩溃了。解决这个问题的直接办法，就是找一块备用的磁盘。一旦有磁盘损坏，其他磁盘立即校验出损坏的数据，立即写到备用磁盘上。写完之后，阵形就恢复原样了，就没有顾虑了。但是必须保证在其他磁盘全力校验恢复数据的过程中，不可再有第二块磁盘损坏，不然便会玉石俱焚！

想到这里，七星开始了他的实验，并且取得成功。就是在整个阵式中增加一块热备盘，平时这块磁盘并不参与组阵，只是在旁边观战，什么也不干。一旦阵中某个人受伤不能参战了，这个热备盘立即顶替他，其他人再把功力传授给他。传授完毕后，就像原来的阵式一样了。如果在大家传授功力时，有IO请求这块损坏磁盘上的数据，那么大家就暂停传授，先应付外来的敌人。当没有来针对这块损坏磁盘的挑衅时，大家会继续传授。

七星的经验不断丰富，他知道不能急躁，所以实验成功之后，七星并没有马上通知那位大侠，而是继续在想有没有可以改进的地方。他想，热备盘平时不参与组阵，那就不能称作阵式的一部分，而是被排斥在外。这块磁盘平时也没有IO，起不到作用，这样就等于浪费了一块磁盘。那么是不是考虑也让它参与到阵形中来呢？如果要参与进来，那让

它担任什么角色呢？热备角色？如果没有人受伤，这个角色在阵中只会是个累赘。怎么办好呢？七星忽然掠过一丝想法，是否可以让阵中各个角色担待一下，从各自的领地保留出一块空间，用作热备盘的角色呢？

这样把热备盘分布在各个磁盘上，就不会形成累赘，并且同时解决了热备盘和大家不协调的问题。说干就干，七星给那位大侠写了一封信，信中称这种阵式为RAID 5E。七星继续琢磨RAID 5E，让阵中每个人都保留一块领地，而不横割，虽然可以做到数据的及时备份，但是这块领地总显得不伦不类。七星突然想到被撕碎的纸片飘然落下的情景，忽然计上心头！既然校验盘都可以横分竖割的融合到数据盘，为什么热备盘不能呢？一样可以！七星想到这里，于是又给那位侠客去了一封信，信中描述这种新的阵式为RAID 5EE。

那位侠客给七星回了一封信：

七星转，北斗移，
英雄无谓千万里。
待到再次相见时，
白发苍，叙知己！

七星看后老泪纵横，颇为感动。相见恨晚啊，到了晚年才遇到人生知己！

4.1.7 RAID 6阵式

如今，七星已经从一个壮小伙变成了个孤苦伶仃的老头。回首过去，从RAID 0一直到RAID 5（E、EE）创立了6种阵式，各种阵式各有所长。他最得意的恐怕就是中庸派的中庸之道。可就是这个中庸之道，却还有一个一直也未能解决的问题，那就是其中任何一种阵式，都最多

同时允许损坏一块磁盘，如果同时损坏多块，整个阵不攻自破！七星想到这里就一阵酸楚。已经是白发苍苍的七星老侠，决定要用晚年最后一点精力来攻破这个难题。

七星老侠一生精研阵式，有很多宝贵的经验。他这次采用了逆向思维，假设这个模型已经做好，然后从逆向分析它是怎么作用的。先描绘出多种模型，然后一个一个地去攻破，找出最适合的模型。

七星描绘了这么一个模型，假设有5块盘组成一个RAID阵列，4块数据盘，一块校验盘，那么同一时刻突然4块数据盘中的两块损坏作废，只剩下两块数据盘和一块校验盘。假设有某种算法，可以恢复这两块丢失磁盘上的数据，那么怎么从这个模型推断出，这个算法是怎么把丢失的两块盘数据都恢复出来的呢？七星冥思苦想。七星在幼年学习方程的时候，知道要求解一个未知数，只需知道只包含这一个未知数的一个等式即可逆向求解。就像布尔的逻辑算式用在RAID 3阵式时候一样，各个数据盘上的数据互相XOR之后就等于校验盘上的校验数据，这就是一个等式：

$$D1 \text{ XOR } D2 \text{ XOR } D3 = \text{Parity}$$

如果此时D1未知，而其他3个值都已知，那么就可以逆向解出未知数，而这也是RAID 3进行校验恢复的时候所做的。那么此时如果D1和D2都未知，也就是1号盘和2号盘都损坏了，还能解出这两个值么？数学告诉七星，这是不可能的，除非除了这个等式还额外存在一个和这个等式不相关的另一个等式！

要求解两个未知数，只要知道关于这两个未知数的不相关的两个关系方程即可。比如有等式为 $D1 \text{ ? } D2 \text{ ? } D3 = *$ ，联立以上两个等式，即

可求出D1和D2。七星开始寻觅这个等式，这个等式是已经存在？还是需要自己去发明呢？七星一开始打算从布尔等式找寻出第二个等式的蛛丝马迹，但是后来他根据因果率知道如果从布尔等式推出其他某些等式，那么推出的等式和布尔等式就是相关的。互相相关的两个等式，在数学上是等价的，无法作为第二个等式。七星有所察觉了，他认为要想得出第二个等式，必须由自己发明一套算法，一套和布尔等式不相关的算法！他开始在纸上演义算法，首先他开始从算术的加减方程开始着手，他写出了可以求解两个未知数的二元方程：

$$\begin{aligned}x+y&=10 \\ 2x+3y&=20\end{aligned}$$

这算是最简单的算术方程了。可以求得 $x=10$ ， $y=0$ 。

以上是对于算术运算的方程，那么布尔逻辑运算是否也可以有方程呢？七星写下如下的式子：

$$\begin{aligned}x \text{ XOR } y &= 1 \\ Ax \text{ XOR } Bx &= 0\end{aligned}$$

第一个方程已经存在了，也就是用在RAID 3上的校验方程。第二个方程是七星模仿加减方程来写的，也就是给 x 和 y 两个值分别加上一个系数，而这两个系数不能是从第一个等式推得的，比如是将第一个等式未知数的系数同乘或同除以某个数得出来的，这样就是相关等式了。七星立即找来布尔逻辑运算方面的书深入学习，终于得到了印证，这种方程确实存在！七星激动得跳了起来！他立即投入到研发中。过了两个月，终于得出了结果！大获成功！

七星对一份数据使用两套算法各自算出一个等式，1号等式右边的

结果写入校验盘1、2号等式右边的值写入校验盘2。这样，只要使用中
有两个值发生丢失，就可以通过这两个等式联立，解出丢失的两个值。
不管这两个值是等式左边的还是等式右边的，只要代入这两个等式中，
就可以求出解。

数学的力量是伟大的，任何东西只要通过了数学的验证，就是永恒的！

同样，七星将用在RAID 5中的方法，用在了新创立的阵式中，将校
验盘分布到数据盘中，不同的是新阵式的校验盘有两块，在每块磁盘上
放置两个等式需要的校验值。

七星给这种阵式取名为RAID 6。

RAID 6相对其他各种中庸派的阵式安全多了，但同时它的写性能更
差了，因为它要多读出一个校验数据，而且计算后还要写入一次，这就
比RAID 5每次写耗费多了两个操作，变成了6次操作，所谓的“写惩
罚”更大了。确实是鱼与熊掌不能兼得啊！正可谓：

寻寻觅觅终冷清，
七星北斗伴我行。
世间万物皆规律，
求得心法谋太平！

七星在创立了RAID 6阵式之后，已经老态龙钟，疾病缠身，所剩时
日不多。而江湖上可是一派热闹，修炼的修炼，修炼好的就立门派，开
设数据库、网站等服务来大赚钱财。而打着七星大侠旗号到处招摇撞骗
的人也不在少数。张真人路见不平拔刀相助，他召开武林大会，表彰了
七星的功绩，指出真正的七星现在早已归隐，那些招摇撞骗的人也就没

有容身之地了。他还提议将七星所创立的7种阵式命名为“七星北斗阵”，以纪念他和七星之间的北斗豪情！

张真人亲自到深山去探望已经下不了床的七星，并将武林中发生的事情告诉了他。此后没几天，七星无憾地离开了人世，化作了北斗七星，在天上洒下无限的光芒照耀世间！张真人给老人办了后事，并将七星过世的消息宣布了出去。没想到第二天，七星、星七、北斗、斗北、七星北斗、北斗七星、星七北斗、星七斗北等商标就被抢注了！大批的商人在发着横财。正可谓：

七星赞

七星阵里论七星，
北斗光前参北斗。
不知天上七星侠，
如今过活要饭否？

4.2 七种RAID技术详解

下面从纯技术角度，深入剖析目前存在的七种RAID模式的组成原理和结构，并分析各种级别相对于单盘IO速率的变化。

首先澄清一点，所谓Stripe完全是由程序在内存中虚拟出来的，说白了就是一个map公式。即仿佛是给程序戴了一个特殊的眼镜，程序戴上这个眼镜，就能看到“条”和“带”，就会知道将数据分布到条带上了。一旦摘下这个眼镜，那么看到的就是普通的物理磁盘扇区。这个眼镜就是实现RAID的程序代码。物理磁盘上根本不存在什么“条”和“带”，只有扇区。另外，程序会在磁盘特定的一些扇区中写入自己运行时需要的信息，比如一些RAID标签信息等。

图4-7为一个RAID 0系统的示意图。

图4-7 一个典型的RAID 0系统

1. 扇区、块、段（Segment）、条带、条带长度和深度

图4-7中的5个竖条，分别代表5个磁盘。然后在磁盘相同偏移处横向逻辑分割，形成Stripee。一个Stripee横跨过的扇区或块的个数或字节容量，就是条带长度，即Stripee Length。而一个Stripee所占用的单块磁盘上的区域，称为一个Segment。一个Segment中所包含的data Block或者扇区的个数或者字节容量，称为Stripee Depth。Data Block可以是N倍个扇区大小的容量，应该可调，或者不可调，由控制器而定。

RAID 0便是将一系列连续编号的Data Block分布到多个物理磁盘上，扩散IO提高性能。其分布的方式如图4-7所示。这个例子中，条带

深度为4，则0、1、2、3号Data Block被放置到第一个条带的第一个Segment中，然后4、5、6、7号Block放置到第一个条带的第二个Segment中，依此类推，条带1放满后，继续放条带2。这种特性称为“局部连续”，因为Block只有在一个Segment中是物理连续的，逻辑连续就需要跨物理磁盘了。

2. 关于几个与IO相关的重要概念

IO可以分为读/写IO、大/小块IO、连续/随机IO、顺序/并发IO、稳定/突发IO、持续/间断IO和实/虚IO。下面来分别介绍这几种IO。

(1) 读/写IO

这个就不用多说了，读IO就是发指令从磁盘读取某段序号连续的扇区的内容。指令一般是通知磁盘开始扇区位置，然后给出需要从这个初始扇区往后读取的连续扇区个数，同时给出动作是读还是写。磁盘收到这条指令就会按照指令的要求读或者写数据。控制器发出这种指令加数据并得到对方回执的过程就是一次IO读或IO写。

注意： 一个IO所要提取的扇区段一定是连续的，如果想提取或写入两段不连续的扇区段，只能将它们放入两个IO中分别执行，这也就是为何随机IO对设备的IOPS指标要求较高的原因。

(2) 大/小块IO

指控制器的指令中给出的连续读取扇区数目的多少。如果数目很大，如128、64等，就应该算是大块IO；如果很小，比如1、4、8等，就应该算是小块IO。大块和小块之间没有明确的界限。

（3）连续/随机IO

连续和随机是指本次IO给出的初始扇区地址和上一次IO的结束扇区地址是不是完全连续的或者相隔不多的。如果是，则本次IO应该算是一个连续IO；如果相差太大，则算一次随机IO。连续IO因为本次初始扇区和上次结束扇区相隔很近，则磁头几乎不用换道或换道时间极短。如果相差太大，则磁头需要很长的换道时间。如果随机IO很多，会导致磁头不停换道，效率大大降低。

（4）顺序/并发IO

意思是，磁盘控制器如果可以同时对一个RAID系统中的多块磁盘同时发送IO指令（当然这里的同时是宏观的概念，如果所有磁盘都在一个总线或者环路上，则这里的同时就是指向一块磁盘发送一条指令后不必等待它回应，接着向另一块磁盘发送IO指令），并且这些最底层的IO数据包含了文件系统级下发的多个IO的数据，则为并发IO。如果这些直接发向磁盘的IO只包含了文件系统下发的一个IO的数据，则此时为顺序IO，即控制器缓存中的文件系统下发的IO队列，只能一个一个来。并发IO模式在特定的条件下可以很大程度地提高效率和速度。

（5）持续/间断IO

持续不断地发送或者接受IO请求数据流，这种情况为持续IO；IO数据流时断时续则为间断IO。

（6）稳定/突发IO

某存储设备或者某程序在一段时间内接收或者发送的IOPS以及Throughput（吞吐量）保持相对稳定和恒定，则称为稳定IO；如果单位

时间的IOPS或者Throughput突然猛增，则为突发IO。

（7）实/虚IO

某IO请求中包含对应实际数据地址的，比如磁盘LBA地址，或者文件偏移量，请求读或者写实际文件或者磁盘扇区数据的，称为实IO；而应用程序针对文件元数据操作的（在文件系统层以上没有文件主体数据操作），或者针对磁盘发送的非实体数据IO请求，比如Report LUN、SCSI Sense Request等控制性IO，称为虚IO。

（8）IO并发几率

单盘，IO并发几率为0，因为一块磁盘同时只可以进行一次IO。对于RAID 0，在2块盘情况下，条带深度比较大的时候（条带太小不能并发IO），并发两个IO的几率为1/2。其他情况请自行运算。

（9）IOPS

完成一次IO所用的时间=寻道时间+旋转延迟时间+数据传输时间， $IOPS = IO\text{并发系数} / (\text{寻道时间} + \text{旋转延迟时间} + \text{数据传输时间})$ 。由于寻道时间相对于传输时间要大几个数量级，所以影响IOPS的关键因素就是降低寻道时间。在连续IO的情况下，寻道时间很短，仅在换磁道时候需要寻道。在这个前提下，传输时间越少，IOPS就越高。

（10）每秒IO吞吐量

显然，每秒IO吞吐量=IOPS×平均IO SIZE。IO SIZE越大，IOPS越高，每秒IO吞吐量就越高。设磁头每秒读写数据速度为V，V为定值。则 $IOPS = IO\text{并发系数} / (\text{寻道时间} + \text{旋转延迟时间} + IO\text{ SIZE}/V)$ 。代入得每秒IO吞吐量=IO并发系数×IO SIZE×V/（V×寻道时间+V×旋转延

迟时间+IO SIZE)。可以看出影响每秒IO吞吐量的最大因素就是IO SIZE和寻道时间以及旋转延迟时间。IO SIZE越大，寻道时间越小，吞吐量越高。相比能显著影响IOPS的因素只有一个，就是寻道时间。

4.2.1 RAID 0技术详析

RAID 0是这样一种模式：我们拿5块盘的RAID 0为例子，如图4-7所示。

对外来说，参与形成RAID 0的各个物理盘会组成一个逻辑上连续、物理上也连续的虚拟磁盘。一级磁盘控制器（指使用这个虚拟磁盘的控制器，如果某台主机使用适配卡链接外部盘阵，则指的就是主机上的磁盘控制器）对这个虚拟磁盘发出的指令，都被RAID控制器收到并分析处理，根据Block映射关系算法公式转换成对组成RAID 0的各个物理盘的真实物理磁盘IO请求指令，收集或写入数据之后，再提交给主机磁盘控制器。

图4-8为一个RAID 0虚拟磁盘的示意图。

图4-8 虚拟磁盘

RAID 0还有另一种非条带化模式，即写满其中一块物理磁盘之后，再接着写另一块，直到所有组成的磁盘全部写满。这种模式对IO写没有任何优化，但是对IO读能提高一定的并发IO读几率。

下面我们来具体分析一个从上到下访问RAID 0磁盘的过程。

假如某一时刻，主机控制器发出指令：读取 初始扇区10000长度128。

RAID控制器接受到这个指令之后，立即进行计算，根据对应公式（这个公式是RAID控制器在做逻辑条带化的时候制定的）算出10000号逻辑扇区所对应的物理磁盘的扇区号。

依次计算出逻辑上连续的下128个扇区所在物理磁盘的扇区号。

分别向对应这些扇区的磁盘再次发出指令。这次是真实的读取数据了，磁盘接受到指令，各自将数据提交给RAID控制器，经过控制器在Cache中的组合，再提交给主机控制器。

分析以上过程，发现如果这128个扇区都落在同一个Segment中的话，也就是说条带深度容量大于128个扇区的容量（64KB），则这次IO就只能真实地从这一块物理盘上读取，性能和单盘相比会减慢，因为没有任何优化，反而还增加了RAID控制器额外的计算开销。所以，在某种特定条件下要提升性能，让一个IO尽量扩散到多块物理盘上，就要减小条带深度。在磁盘数量不变的条件下，也就是减小条带大小（Stripe SIZE，也就是条带长度）。让这个IO的数据被控制器分割，同时放满一个条带的第一个Segment、第二个Segment等，依此类推，这样就能极大地占用多块物理盘。

误区：总是以为控制器是先放满第一个Segment，再放满第二个Segment。

其实是同时进行的，因为控制器把每块盘要写入或者读取的数据都计算好了。如果这些目标磁盘不在相同的总线中，那么这种宏观“同时”的粒度将会更加细。因为毕竟计算机总线是共享的，一个时刻只能对一个外设进行IO。

所以，RAID 0要提升性能，条带做的越小越好。但是又一个矛盾出

现了，就是条带太小，导致并发IO几率降低。因为如果条带太小，则每次IO一定会占用大部分物理盘，队列中的IO就只能等待这次IO结束后才能使用物理盘。而条带太大，又不能充分提高传输速度。这两个是一对矛盾，要根据需求来采用不同的方式。如果随机小块IO多，则适当加大条带深度；如果连续大块IO多，则适当减小条带深度。

接着分析RAID 0相对于单盘的性能变化。根据以上总结出来的公式，可以推出表4-1。

表4-1 RAID 0系统相对于单盘的IO对比

注：并发IO和IO SIZE/Stripe SIZE是一对矛盾，两者总是对立的。N=组成RAID 0的磁盘数目。
系数=IO SIZE/Stripe SIZE和初始LBA地址所处的Stripe偏移综合系数，大于等于1。并发系数=并发IO的数量。

4.2.2 RAID 1技术详析

RAID 1是这样一种模式：拿两块盘的例子来进行说明，如图4-9所示。

图4-9 RAID 1系统示意图

RAID 1和RAID 0不同，RAID 0对数据没有任何保护措施，每个Block都没有备份或者校验保护措施。RAID 1对虚拟逻辑盘上的每个物理Block，都在物理盘上有一份镜像备份，也就是说数据有两份。对于RAID 1的写IO，速度不但没有提升，而且有所下降，因为数据要同时向多块物理盘写，时间以最慢的那个为准，因为是同步的。而对于RAID 1的读IO请求，不但可以并发，而且就算顺序IO的时候，控制器也可以像RAID 0一样，从两块物理盘上同时读数据，提升速度。RAID

1可以没有Stripe的概念，当然也可以有，同样可总结出表4-2。

表4-2 RAID 1系统相对于单盘的IO对比

注： N = 组成RAID 1镜像物理盘的数目。

在读、并发IO的模式下，由于可以并发 N 个IO，每个IO占用一个物理盘，这就相当于提升了 N 倍的IOPS。由于每个IO只独占了一个物理盘，所以数据传输速度相对于单盘并没有改变，所以不管是随机还是顺序IO，相对单盘都不变。

在读、顺序IO、随机IO模式下，由于IO不能并发，所以此时一个IO可以同时读取 N 个盘上的内容。但是在随机IO模式下，寻道时间影响很大，纵使同时分块读取多个磁盘的内容，也架不住寻道时间的抵消，所以性能提升极小。

在读、顺序IO、连续IO模式下，寻道时间影响最低，此时传输速率为主要矛盾，同时读取多块磁盘的数据，时间减少为 $1/N$ ，所以性能提升了 N 倍。

写IO的时候和读IO情况相同，就不做分析了。写IO因为要同时向每块磁盘写入备份数据，所以不能并发IO，也不能分块并行。但是如果控制器把优化算法做到极致的话，还是可以并发IO的，比如控制器从IO队列中提取连续的多个IO，可以将这些IO合并，并发写入磁盘，前提是这几个IO必须是事务性的，也就是说LBA必须连续，不然不能作为一个大的合并IO。而且和文件系统也有关系，文件系统碎片越少，并发几率越高。

4.2.3 RAID 2技术详析

RAID 2是一种比较特殊的RAID模式，它是一种专用RAID，现在早已被淘汰。它的基本思想是在IO到来之后，控制器将数据按照位分散开，顺序在每块磁盘中存取1b。这里有个疑问，磁盘的最小IO单位是扇区，有512B，如何写入1b？其实这个写入1b，并非只写入1b。我们知道上层IO可以先经过文件系统，然后才通过磁盘控制器驱动来向磁盘发出IO。最终的IO大小，都是N倍的扇区，也就是 $N \times 512B$ ，N大于等于1，不可能发生N小于1的情况。即使需要的数据只有几个字节，那么也同样要读出或写入整个扇区，也就是512B。

明白这个原则之后，再来看一下RAID 2中所谓的“每个磁盘写1b”是个什么概念。IO最小单位为扇区（512B），我们就拿一个4块数据盘和3块校验盘的RAID 2系统为例给大家说明一下。这个环境中，RAID 2的一个条带大小是4b（1b×4块数据盘），而IO最小单位是一个扇区，那么如果分别向每块盘写1b，就需要分别向每块盘写一个扇区，每个扇区只包含1b有效数据，这显然是不可能的，因为太浪费空间，且没有意义。

下面以IO请求为例来说明。

写入“初始扇区10000长度1”，这个IO目的是要向LBA10000写入一个扇区的数据，也就是512B。

RAID 2控制器接受到这512B的数据后，在Cache中计算需要写入的物理磁盘的信息，比如定位到物理扇区，分割数据成比特。

然后一次性写入物理磁盘扇区。也就是说第一块物理盘，控制器会写入本次IO数据的第1、5、9、13、17、21等位，第二块物理盘会写入2、6、10、14、18、22等位，其他两块物理盘同样方式写入。

直到这样将数据写完。我们可以计算出来，这512B的数据写完之

后，此时每块物理盘只包含128B的数据，也就是一个扇区的四分之一，那么这个扇区剩余的部分，就是空的。

为了利用起这部分空间，等下次IO到来之后，控制器会对数据进行比特分割，将数据填入这些空白区域。控制器将首先读出原来的数据，然后和新数据合并之后，一并再写回这个扇区，这样做效率和速度都大打折扣。其实RAID 2就是将原本连续的一个扇区的数据，以位为单位，分割存放到不连续的多块物理盘上，因为这样可以在任意条件下都迫使其全磁盘组并行读写，提高性能，也就是说条带深度为1位。这种极端看上去有点做得过火了，这也是导致它最终被淘汰的原因之一。

RAID 2系统中每个物理磁盘扇区其实是包含了N个扇区的“残体”。

思考：那么如果出现需要更新这4个扇区中某一个扇区的情况，怎么办？

这种情况下，必须先读出原来的数据，和新数据合并，然后再一并写入。其实这种情况出现的非常少。我们知道上层IO的产生，一般是需要先经过OS的文件系统，然后才到磁盘控制器这一层的。所以磁盘控制器产生的IO一般都是事务性的，也就是这个IO中的所有扇区很大几率上对于上层文件系统来说是一个完整的事务，所以很少会发生只针对这个事务中某一个点进行读写的情况。

这样的话，每次IO就有很大概率都会包含入这些逻辑上连续的扇区，所以不必担心经常会发生那种情况。即便发生了，控制器也只能按照那种低效率的做法来做，不过总体影响较小。但是如果随机IO比较多，那么这些IO初始LBA，很有可能就会命中在一个两个事务交接的扇区处。这种情况就会导致速度和效率大大降低。连续IO出现这种情况的

几率非常小了。

RAID 2因为每次读写都需要全组磁盘联动，所以为了最大化其性能，最好保证每块磁盘主轴同步，使同一时刻每块磁盘磁头所处的扇区逻辑编号都一致，并存并取，达到最佳性能。如果不能同步，则会产生等待，影响速度。

基于RAID 2并存并取的特点，RAID 2不能实现并发IO，因为每次IO都占用了每块物理磁盘。

RAID 2的校验盘对系统不产生瓶颈，但是会产生延迟，因为多了计算校验的动作。校验位和数据位是一同并行写入或者读取的。RAID 2采用汉明码来校验数据，这种码可以判断修复一位错误的的数据，并且使用校验盘的数量太多，4块数据盘需要3块校验盘。但是随着数据盘数量的增多，校验盘所占的比例会显著减小。

RAID 2和RAID 0有些不同，RAID 0不能保证每次IO都是多磁盘并行，因为RAID 0的条带深度相对于RAID 2以位为单位来说是太大了。而RAID 2由于每次IO都保证是多磁盘并行，所以其数据传输率是单盘的N倍。为了最好地利用这个特性，就需要将这个特性的主导地位体现出来。

而根据 $IOPS = IO \text{ 并发系数} / (\text{寻道时间} + \text{旋转延迟时间} + \text{数据传输时间})$ ，寻道时间比数据传输时间要大几个数量级。所以为了体现数据传输时间减少这个优点，就必须避免寻道时间的影响，而最佳做法就是尽量产生连续IO而不是随机IO。所以，RAID 2最适合连续IO的情况。另外，根据 $\text{每秒IO吞吐量} = IO \text{ 并发系数} \times IO \text{ SIZE} \times V / (V \times \text{寻道时间} \times V + \text{旋转延迟时间} + IO \text{ SIZE})$ ，如果将IO SIZE也增大，则每秒IO吞吐量也

将显著提高。所以，RAID 2最适合的应用就是产生连续IO、大块IO的情况。不言而喻，视频流服务等应用适合RAID 2。不过，RAID 2的缺点太多，比如校验盘数量多、算法复杂等，它逐渐被RAID 3替代了。表4-3比较了RAID 2系统与单盘的性能。

表4-3 RAID 2系统相对于单盘的IO对比

注： N = 数据盘数量。RAID 2不能并发IO。

4.2.4 RAID 3技术详析

图4-10所示为一个RAID 3系统的条带布局图。

图4-10 RAID 3系统示意图

RAID 2缺点比较多，比如非事务性IO对它的影响、校验盘数量太多等。RAID 2的劣势就在于它将数据以比特为单位进行分割，将原本物理连续的扇区转变成物理不连续，而逻辑连续的。这样就导致了它对非事务性IO的效率低下。为了从根本上解决这个问题，RAID 3出现了。

既然要从根本上解决这个问题，首先就是需要抛弃RAID 2对扇区进行分散的做法，RAID 3保留了扇区的物理连续。RAID 2将数据以比特为单位分割，这样是为了保证每次IO占用全部磁盘的并行性。而RAID 3同样也保留了这个特点，但是没有以比特为单位来分散数据，而是以一个扇区或者几个扇区为单位来分散数据。RAID 3还采用了高效的XOR校验算法，但是这种算法只能判断数据是否有误，不能判断出哪一位有误，更不能修正错误。XOR校验使得RAID 3可以不管多少块数据盘，只需要一块校验盘就足够了。

RAID 3的每一个条带，其长度被设计为一个文件系统块的大小，深

度随磁盘数量而定，但是最小深度为1个扇区。这样的话，每个Segment的大小一般就是1个扇区或者几个扇区的容量。以图4.10的例子来看，有4块数据盘和1块校验盘。每个Segment也就是图中的一个Block Portion，假如为两个扇区大小，就是1KB，则整个条带的数据部分大小为4KB。如果一个Segment大小为8个扇区，即4KB，则整个条带大小为16KB。

例解：RAID 3的作用机制

还是用一个例子来说明RAID 3的作用机制。一个4块数据盘和1块校验盘的RAID 3系统，Segment SIZE为两个扇区大小（1KB），条带长度为4KB。

RAID 3控制器接收到了这样一个IO：“写入 初始扇区10000 长度8”，即总数据量为 $8 \times 512\text{B} = 4\text{KB}$ 。

控制器先定位LBA10000所对应的真实物理LBA，假如LBA10000恰好在第一个条带的第一个Segment的第一个扇区上，那么控制器将这个IO数据里的第1、2个512B写入这个扇区。

同一时刻，第3、4个512B会被同时写入这个条带的第二个Segment中的两个扇区，其后的数据同样被写入第3、4个Segment中，此时恰好是4KB的数据量。也就是说这4KB的IO数据同时被分散写入了4块磁盘，每块磁盘写入了两个扇区，也就是一个Segment。它们是并行写入的，包括校验盘也是并行写入的，所以RAID 3的校验盘没有瓶颈，但是有延迟，因为增加了计算校验的开销。

但现代控制器一般都使用专用的XOR硬件电路而不是CPU来计算

XOR，这样就使得延迟降到最低。上面那种情况是IO SIZE刚好等于一个条带大小的时候，如果IO SIZE小于一个条带大小呢？

还是刚才那个环境，此时控制器接收到IO大小为2KB的写入请求，也就是4个连续扇区，那么控制器就只能同时写入两个磁盘了，因为每个盘上的Segment是两个扇区，也只能得到两倍的单盘传输速率。同时为了更新校验块，写惩罚也出现了。但是如果同时有个IO需要用到另外两块盘，那么恰好可以和当前的IO合并起来，这样就可以并发IO，这种相邻的IO一般都是事务性的连续IO。

再看看IO SIZE大于条带长度的情况。还是那个环境，控制器收到的IO SIZE为16KB。则控制器一次所能并行写入的是4KB，这16KB就需要分4批来写入4个条带。其实这里的分4批写入，不是先后写入，而是同时写入，也就是这16KB中的第1、5、9、13KB将由控制器连续写入磁盘1，第2、6、10、14KB，连续写入磁盘2，依此类推。直到16KB数据全部写完，是并行一次写完。这样校验盘也可以一次性计算校验值并且和数据一同并行写入，而不是“分批”。

通过比较，我们发现，与其使IO SIZE小于一个条带的大小，从而空闲出一些磁盘，不如使IO SIZE大于或者等于条带大小，取消磁盘空余。因为上层IO SIZE是不受控的，控制器说了不算，但是条带大小是控制器说了算的。所以如果将条带大小减到很小，比如两个扇区、一个扇区，则每次上层IO一般情况下都会占用所有磁盘进行并发传输。这样就可以提供和RAID 2一样的传输速度，并避免RAID 2的诸多缺点。RAID 3和RAID 2一样不能并发IO，因为一个IO要占用全部盘，就算IO SIZE小于Stripe SIZE，因为校验盘的独享也不能并发IO。

思考： 一般来说，RAID3的条带长度=文件系统块大小。这

样，就不会产生条带不对齐的现象，从而避免产生碎片。

虽然纯RAID 3系统不能并发IO，但是可以通过巧妙的设计，形成RAID 30系统。如果文件系统块为4KB，则使用8块数据盘+2块校验盘做成的RAID 30系统，便可以并发2个IO了。表4-4比较了RAID 3系统与单盘的性能。

表4-4 RAID 3系统相对于单盘的IO对比

注： N = 组成RAID 3的数据磁盘数量。和RAID 2相同，事务性连续IO可能并发。

和RAID 2一样，RAID 3同样也是最适合连续大块IO的环境，但是它比RAID 2成本更低，更容易部署。

不管任何形式的RAID，只要是面对随机IO，其性能与单盘比都没有大的优势，因为RAID所做的只是提高传输速率、并发IO和容错。随机IO只能靠降低单个物理磁盘的寻道时间来解决。而RAID不能优化寻道时间。所以对于随机IO，RAID 3也同样没有优势。

而对于连续IO，因为寻道时间的影响因素可以忽略，RAID 3最拿手了。因为像RAID 2一样，RAID 3可以大大加快数据传输速率，因为它是多盘并发读写。所以理论上可以相当于单盘提高 N 倍的速率。

但是RAID 3最怕的就是遇到随机IO。由于在RAID 3下，每个IO都需要牵动所有盘来为它服务，这样的话，如果向RAID 3组发送随机IO，那么所有磁盘均会频繁寻道，此时，整个RAID 组所表现出来的性能甚至不如单盘，因为就算组内有100块盘，由于随机IO的到来，所有磁盘均忙于寻道，100块盘并发寻道所耗费的时间与一块盘是相同的。

4.2.5 RAID 4技术详析

图4-11是一个RAID 4系统的条带布局图。

图4-11 RAID 4系统示意图

不管是RAID 2还是RAID 3，它们都是为了大大提高数据传输率而设计，而不能并发IO。诸如数据库等应用的特点就是高频率随机IO读。想提高这种环境的IOPS，根据公式 $IOPS = IO \text{并发系数} / (\text{寻道时间} + \text{旋转延迟时间} + \text{数据传输时间})$ ，随机读导致寻道时间和旋转延迟时间增大，靠提高传输速率已经不是办法。所以观察这个公式，想在随机IO频发的环境中提高IOPS，唯一能够做的是要么用高性能的磁盘（即平均寻道时间短的磁盘），要么提高IO并发系数。不能并发IO的，想办法让它并发IO。并发系数小的，想办法提高系数。

思考：在RAID 3的基础上，RAID 4被发展起来。我们分析RAID 3的性能的时候，曾经提到过一种情况，就是IO SIZE小于Stripe SIZE的时候，此时有磁盘处于空闲状态。如果抓住这个现象，同时让队列中的其他IO来利用这些空闲的磁盘，岂不是正好达到并发IO的效果了么？所以RAID 4将一个Segment的大小做得比较大，以至于平均IO SIZE总是小于Stripe SIZE，这样就能保证每个IO少占用磁盘，甚至一个IO只占用一个磁盘。

是的，这个思想对于读IO是对路子的，但是对于写IO的话，有一个很难克服的问题，那就是校验盘的争用。考虑一下这样一种情况，在4块数据盘和1块校验盘组成的RAID 4系统中，某时刻一个IO占用了前两块盘和校验盘，此时虽然后两块是空闲的，可以同时接受新的IO请求。但是接受了新的IO请求，则新IO请求同样也要使用校验盘。由于一块物理磁盘不能同时处理多个IO，所以新IO仍然要等旧IO写完后，才能写入校验。这样就和顺序IO没区别了。数据盘可并发而校验盘不可并发，这

样不能实现写IO并发。

如果仅仅根据争用校验盘来下结论说RAID 4不支持并发IO，也是片面的。我们可以设想这样一种情形，某时刻一个IO只占用了全部磁盘的几块盘，令一些磁盘空闲。如果此时让队列中下一个IO等待的话，那么当然不可能实现并发IO。

思考：如果队列中有这样一个IO，它需要更新的LBA目标和正在进行的IO恰好在同一条带上，并且处于空闲磁盘，还不冲突，那么此时我们就可以让这个IO也搭一下正在进行的IO的顺风车。反正都是要更新这个条带的校验Segment，与其两个IO先后更新，不如让它们同时更新各自的数据Segment，而控制器负责计算本条带的校验块。这样就完美的达到了IO并发。

但是，遇到这种情况的几率真是小之又小。即便如此，控制器如果可以对队列中的IO目标LBA进行扫描，将目标处于同一条带的IO，让其并发写入，这就多少类似NCQ技术了。但是如果组合动作在上层就已经算好了，人为的创造并发条件，主动去合并可以并发的，合并好之后再下发给下层，那么事务并发IO的几率将大大增加，而不是靠底层碰运气来实现，不过此时称为“并发事务”更为合适。

所谓“上层”是什么呢？上层就是一级磁盘控制器驱动程序的上层，也就是文件系统层。文件系统管理着底层磁盘，决定数据写往磁盘或者虚拟卷上的哪些块。所以完全可以在文件系统这个层次上，将两个不同事务的IO写操作，尽量放到相同的条带上。比如一个条带大小为16KB，可以前8KB放一个IO的数据，后8KB放另一个IO的数据，这两个IO在经过文件系统的计算之后，经由磁盘控制器驱动程序，向磁盘发出同时写入整个条带的操作，这样就构成了整条写。如果实在不能占满

整条，那么也应该尽量达成重构写模式，这样不但并发了IO，还使得写效率增加。

提示： 这种在文件系统专门为RAID 4做出优化的方案，最典型的的就是NetApp公司著名的WAFL文件系统。WAFL文件系统的设计方式确保了能够最大限度地实现整条写操作。

图4-12对比显示了WAFL如何分配同样的数据块，从而使得RAID 4更加有效。WAFL总是把可以合并写入的数据块尽量同时写到一个条带中，以消除写惩罚，增加IO并发系数。相对于WAFL，左边的FFS（普通文件系统）由于对RAID 4没有感知，产生的IO不适合RAID 4的机制，从而被零散地分配到了6个独立的条带，因此致使6个校验盘块需要更新，而只能顺序的进行，因为校验盘不可并发。而右边的WAFL仅仅使用3道条带，只有3个校验块需要更新，从而大大提高了性能。表4-5比较了RAID 4系统与单盘的性能。

图4-12 普通文件系统与WAFL文件系统的对比

表4-5 RAID 4系统相对于单盘的IO对比

注： *N* 为RAID 4数据盘数量。IO SIZE/Stripe SIZE太大则并发IO几率很小。

注意： 如果IO SIZE/Stripe SIZE的值太小，那么顺序IO读不管是连续还是随机IO几乎都没有提升。顺序IO写性能下降，是因为IO SIZE很小，又是顺序IO，只能进行读改写，性能会降低不少。

所以，如果要使用RAID 4，不进行特别优化是不行的，至少要让它可以进行并发IO。观察表4-5可知，并发IO模式下性能都有所提升。然而如果要优化到并发几率很高，实在不容易。目前只有NetApp的WAFL

文件系统还在使用RAID 4，其他产品均未见使用。RAID 4面临淘汰，取而代之的是拥有高盲并发几率的RAID 5系统。所谓盲并发几率，就是说上层不必感知下层的结构，即可增加并发系数。

4.2.6 RAID 5技术详析

图4-13为一个RAID 5系统的条带布局图。

图4-13 RAID 5系统示意图

先介绍几个概念：整条写、重构写与读改写。

（1）整条写（**Full-Stripe Write**）：需要修改奇偶校验群组中所有的条带单元，因此新的XOR校验值可以根据所有新的条带数据计算得到，不需要额外的读、写操作。因此，整条写是最有效的写类型。整条写的例子，如RAID 2、RAID 3。它们每次IO总是几乎能保证占用所有盘，因此每个条带上的每个Segment都被写更新，所以控制器可以直接利用这些更新的数据计算出校验数据之后，在数据被写入数据盘的同时，将计算好的校验信息写入校验盘。

（2）重构写（**Reconstruct Write**）：如果要写入的磁盘数目超过阵列磁盘数目的一半，可采取重构写方式。在重构写中，从这个条带中不需要修改的Segment中读取原来的数据，再和本条带中所有需要修改的Segment上的新数据一起计算XOR校验值，并将新的Segment数据和没有更改过的Segment数据以及新的XOR校验值一并写入。显然，重构写要牵涉更多的I/O操作，因此效率比整条写低。重构写的例子，比如在RAID 4中，如果数据盘为8块，某时刻一个IO只更新了一个条带的6个Segment，剩余两个没有更新。在重构写模式下，会将没有被更新的两个Segment的数据读出，和需要更新的前6个Segment的数据计算出校验

数据，然后将这6个Segment连同校验数据一并写入磁盘。可以看出，这个操作只是多出了读两个Segment中数据的操作和写Parity校验数据的操作，但是写的时候几乎不产生延迟开销，因为是宏观同时写入。

（3）读改写（**Read-Modify Write**）：如果要写入的磁盘数目不足阵列磁盘数目的一半，可采取读改写方式。读改写过程是：先从需要修改的Segment上读取旧的数据，再从条带上读取旧的奇偶校验值；根据旧数据、旧校验值和需要修改的Segment上的新数据计算出这个条带上的新的校验值；最后写入新的数据和新的奇偶校验值。这个过程中包含读取、修改和写入的一个循环周期，因此称为读改写。读改写计算新校验值的公式为：新数据的校验数据 = （老数据 EOR 新数据） EOR 老校验数据。如果待更新的Segment已经超过了条带中总Segment数量的一半，则此时不适合用读改写，因为读改写需要读出这些Segment中的数据和校验数据。而如果采用重构写，只需要读取剩余不准备更新数据的Segment中的数据即可，而后者数量比前者要少。所以超过一半用重构写，不到一半用读改写，整条更新就用整条写。

写效率排列为：整条写>重构写>读改写。

图4-14是RAID 5系统的三种写模式示意图。

图4-14 RAID 5写模式示意图

为了解决RAID 4系统不能并发IO的窘境，RAID 5相应而出。RAID 4并发困难是因为它的校验盘争用的问题，如果能找到一种机制可以有效地解决这个问题，则实现并发就会非常容易。RAID 5恰恰解决了校验盘争用这个问题。RAID 5采用分布式校验盘的做法，将校验盘打散在RAID组中的每块磁盘上。如图4-13所示，每个条带都有一个校验

Segment，但是不同条带中其位置不同，在相邻条带之间循环分布。为了保证并发IO，RAID 5同样将条带大小做得较大，以保证每次IO数据不会占满整个条带，造成队列中其他IO的等待。所以，RAID 5要保证高并发率，一旦某时刻没有成功进行并发，则这个IO几乎就是读改写模式，所以RAID 5拥有较高的写惩罚。

但是在随机写IO频发的环境下，由于频发的随机IO提高了潜在的并发几率，如果碰巧并发的IO同处一个条带，还可以降低写惩罚的几率。这样，RAID 5系统面对频发的随机写IO，其IOPS下降趋势比其他RAID类型要平缓一些。

来分析一下RAID 5具体的作用机制。以图4-13的环境为例，条带大小80KB，每个Segment大小16KB。

某一时刻，上层产生一个写IO：写入初始扇区10000长度8，即写入4KB的数据。控制器收到这个IO之后，首先定位真实LBA地址，假设定位到了第1个条带的第2个Segment（位于图中的磁盘2）的第1个扇区（仅仅是假设），则控制器首先对这个Segment所在的磁盘发起IO写请求，读取这8个扇区中原来的数据到Cache。

与此同时，控制器也向这个条带的校验Segment所在的磁盘（即图中的磁盘1）发起IO读请求，读出对应的校验扇区数据并保存到Cache。

利用XOR校验电路来计算新的校验数据，公式为：新数据的校验数据 = （老数据 EOR 新数据） EOR 老校验数据。现在Cache中存在：老数据、新数据、老校验数据和新校验数据。

控制器立即再次向相应的磁盘同时发起IO写请求，将新数据写入数据Segment，将新校验数据写入校验Segment，并删除老数据和老校验数

据。

在上述过程中，这个IO占用的始终只有1、2两块盘，因为所要更新的数据Segment对应的校验Segment位于1盘，自始至终都没有用到其他任何磁盘。如果此时队列中有这么一个IO，它的LBA初始目标假如位于图4-13中下面方框所示的数据Segment中（磁盘4），IO长度也不超过Segment的大小。而这个条带对应的校验Segment位于磁盘3上。这两块盘未被其他任何IO占用，所以此时控制器就可以并发的处理这个IO和上方红框所示的IO，达到并发。

RAID 5相对于经过特别优化的RAID 4来说，在底层就实现了并发，可以脱离文件系统的干预。任何文件系统的IO都可以实现较高的并发几率，又称为盲并发。而不像基于WAFL文件系统的RAID 4，需要在文件系统上规划计算出并发环境。然而就效率来说，仍然是WAFL拥有更高的并发系数，因为毕竟WAFL是靠主动创造并发，而RAID 5却是做好了陷阱等人往里跳，抓着一个是一个。

RAID 5磁盘数量越多，可并发的几率就越大。表4-6比较了RAID 5与单盘的性能。

表4-6 RAID 5系统相对于单盘的IO对比

注：RAID 5最适合小块IO。并发IO的情况下，性能都较单盘有所提升。

图4-15为一个RAID 5E系统的条带布局图。

图4-15 RAID 5E系统示意图（HS代表HotSpare）

图4-16则为一个RAID 5EE系统的条带布局图。

图4-16 RAID 5EE系统示意图

4.2.7 RAID 6技术详析

图4-17为一个RAID 6系统的条带布局图。

图4-17 RAID 6系统示意图

RAID 6之前的任何RAID级别，最多能保障在坏掉一块盘的时候，数据仍然可以访问。如果同时坏掉两块盘，则数据将会丢失。为了增加RAID 5的保险系数，RAID 6被创立了。RAID 6比RAID 5多增加了一块校验盘，也是分布打散在每块盘上，只不过是用另一个方程式来计算新的校验数据。这样，RAID 6同时在一个条带上保存了两份数学上不相关的校验数据，这样能够保证同时坏两块盘的情况下，数据依然可以通过联立这两个数学关系等式来求出丢失的数据。RAID 6与RAID 5相比，在写的时候会同时读取或者写入额外的一份校验数据。不过由于是并行同时操作，所以不比RAID 5慢多少。其他特性则和RAID 5类似。

表4-6比较了RAID 6与单盘的性能。

表4-7 RAID 6系统相对于单盘的IO对比

第5章 降龙传说——RAID、虚拟磁盘、卷和文件系统实战

■ RAID卡

■ 软RAID

■ 虚拟磁盘

■ 卷

■ 文件系统

七星大侠将七星北斗阵式永传于世，虽然其思想博大精深，但是并没有给出如何去具体地实现这七种阵式。但没有关系，有了正确的思想才能更好地指导实践。人们根据七星北斗的思想，发明了各种各样的RAID实现方式。

然而，实现了各种RAID，许多问题也随之而来，且看人们是怎么运用各种手段来解决这些问题的。

5.1 操作系统中**RAID**的实现和配置

有人直接在主机上编写程序，运行于操作系统底层，将从主机SCSI或者IDE控制器提交上来的物理磁盘，运用七星北斗的思想，虚拟成各种模式的虚拟磁盘，然后再提交给上层程序接口，如卷管理程序。这些软件通过一个配置工具，让使用者自行选择将哪些磁盘组合起来并形成哪种类型的RAID。

比如，某台机器上安装了两块IDE磁盘和4块SCSI磁盘，IDE硬盘直接连接到主板集成的IDE接口上，SCSI磁盘则是连接到一块PCI接口的SCSI卡上。在没有RAID程序参与的条件下，系统可以识别到6块磁盘，并且经过文件系统格式化之后，挂载到某个盘符或者目录下，供应用程序读写。

安装了RAID程序之后，用户通过配置界面，先将两块IDE磁盘做成一个RAID 0系统。如果原来每块IDE磁盘是80GB容量，做成RAID 0之后就变成了一块160GB容量的“虚拟”磁盘。然后用户又将4块SCSI盘做了一个RAID 5系统，如果原来每块SCSI磁盘是73GB容量，4块盘做成RAID 5之后虚拟磁盘的容量将约为3块盘的容量，即216GB。

当然，因为RAID程序需要使用磁盘上的部分空间来存放一些RAID信息，所以实际容量将会变小。经过RAID程序的处理之后，这6块磁盘最终变成了两块虚拟磁盘。如果是在Windows系统中，打开磁盘管理器只能看到两块硬盘，一块容量为160GB（硬盘1），另一块容量为219GB（硬盘2）。之后，可以对这两块盘进行格式化，比如格式化为NTFS文件系统。格式化程序丝毫不会感觉到有多块物理硬盘正在写入数据。

比如，格式化程序某时刻发出命令，向硬盘1（由两块IDE磁盘组成的RAID 0虚拟盘）的LBA起始地址10000，长度128，写入内存起始地址某某的数据。RAID程序会截获这个命令并做分析，硬盘1是一个RAID 0系统，那么这块从LBA10000开始算起的128个扇区的数据，会被RAID引擎计算，将逻辑LBA对应成物理磁盘的物理LBA，将对应的数据写入物理磁盘。写入之后，格式化程序会收到成功写入的信号，然后接着做下一次IO。经过这样的处理，上层程序完全不会知道底层物理磁盘的细节。其他RAID形式也都是相同的道理，只不过算法更加复杂而已。但是即使再复杂的算法，经过CPU运算，也要比磁盘读写速度快几千几万倍。

提示： 为了保证性能，同一个磁盘组只能用相同类型的磁盘，虽然也可以设计成将 IDE磁盘和SCSI磁盘组合成虚拟磁盘，不过除非特殊需要，否则没有这样设计的。

5.1.1 Windows Server 2003高级磁盘管理

下面以Windows Server 2003企业版操作系统为例，示例一下Windows是如何在操作系统上用软件来实现RAID功能的。

每个例子的环境都是一个具有5块物理磁盘的PC，每块磁盘容量为100MB。

1. 磁盘初始化和转换

（1）新磁盘插入机箱并启动操作系统之后，打开磁盘管理器，Windows会自动弹出一个配置新磁盘的向导，如图5-1所示。

图5-1 初始界面

(2) 单击“下一步”按钮，出现图5-2所示的对话框。

图5-2 选择要初始化的磁盘

(3) 单击“下一步”按钮，初始化所有新磁盘，如图5-3所示。

图5-3 选择要转换的磁盘

(4) 单击“下一步”按钮，将所有磁盘转换为动态磁盘，如图5-4所示。所谓的动态磁盘就是可以用来做RAID以及卷管理的磁盘。

图5-4 初始化磁盘

(5) 单击“完成”按钮。查看磁盘管理器中的状态，如图5-5所示。

图5-5 磁盘状态

我们从图5-5中可以看到，磁盘0为基本磁盘，同时也是系统所在的磁盘以及启动磁盘。这个磁盘不能对其进行软RAID或卷管理操作。

2. 新建卷

在“磁盘1”上右击，在弹出的快捷菜单中选择“新建卷”命令，如图5-6所示，系统弹出“新建卷向导”对话框，以选择要创建的卷的类型，如图5-7所示。

图5-6 选择“新建卷”命令

图5-7 选择卷类型

这里有5个选项，下面分别介绍。

- 简单卷：指卷将按照磁盘的顺序依次分配空间。简单卷与磁盘分区功能类似，卷空间只能在一块磁盘上分配，并且不能交叉或者乱序。
- 跨区卷：跨区卷在简单卷的基础上，可以让一个卷的空间跨越多块物理磁盘。相当于不做条带化的RAID 0系统。
- 带区卷：带区卷相当于条带化的RAID 0系统。
- 镜像卷：镜像卷相当于RAID 1系统。
- RAID-5卷：毫无疑问，这种方式就是实现一个RAID 5卷。

图5-8做的是一个大小为101MB的简单卷，也就是将物理磁盘1全部容量划分给这个卷。可以发现，简单卷只能在一块物理磁盘上划分，图中“添加”按钮是灰色的，证明不能跨越多块磁盘。

图5-8 划分大小

我们再来看看跨区卷，如图5-9所示。

图5-9 跨区卷

跨区卷允许卷容量来自多个硬盘，并且可以在每个硬盘上选择部分容量而不一定非要选择全部容量。在此，我们将全部容量划分给这个卷，卷总容量为200MB，如图5-10所示。

图5-10 跨区卷状态

建好的跨区卷，将用紫色来表示。此外，还可以灵活地扩展这个卷的容量，如图5-11所示。

图5-11 扩展容量

向这个卷中再添加一块磁盘“磁盘3”，如图5-12所示。

图5-12 增加一块物理磁盘

加完之后这个卷的容量就被扩充到了300MB，如图5-13所示。

图5-13 扩容后的卷

3. 删除卷

如图5-14所示，可以任意删除卷。

图5-14 删除卷

下面用磁盘1的前50MB的容量和磁盘2的全部容量来做一个跨区卷，如图5-15所示。

图5-15 灵活地划分尺寸

做好后的卷如图5-16所示。此外，磁盘1剩余的51MB容量还可以再新建卷，如图5-16所示。

图5-16 剩余空间可以新建卷

4. 带区卷

下面我们来做一个带区卷，即条带化的RAID 0卷，选择用磁盘1和磁盘2中各30MB的容量来做一个60MB的卷，如图5-17和图5-18所示。

图5-17 带区卷

图5-18 带区卷的状态

做好之后的带区卷会用绿色标识。

5. 镜像卷

我们再来做一个镜像卷，即RAID 1卷，选择用磁盘1和磁盘2中各40MB的容量来做一个40MB的卷，如图5-19和图5-20所示。

图5-19 镜像卷

图5-20 镜像卷的状态

做好后的镜像卷会用棕色标识。

6. RAID 5类型的卷

最后，我们来做一個RAID 5类型的卷，可将所有磁盘的各50MB空间做一个卷，如图5-21所示；然后再用所有硬盘的20MB空间做一个卷，形成两个RAID 5卷。

图5-21 创建RAID 5卷

做好后的RAID 5卷会用亮绿色标识，如图5-22所示。

图5-22 RAID 5卷的状态

提示：做好的任何卷均可随意被删除，如图5-23所示。

图5-23 删除了一个RAID 5卷

说明： Windows的动态磁盘管理实际上应该算是一个带有RAID功能的卷管理软件，而不仅仅是RAID软件。卷管理的概念我们在下文会解释。

5.1.2 Linux下软RAID配置示例

下面在一台装有8块物理磁盘的机器上安装RedHat Enterprise Linux Server 4 Update 5操作系统，具体操作过程如下

(1) 选择手动配置磁盘界面，如图5-24所示。

图5-24 选择手动配置

(2) 可以看到系统识别到了8块物理磁盘，如图5-25所示。

图5-25 识别到的磁盘列表

(3) 必须划分一个/boot分区用来启动基本的操作系统内核。用第一块磁盘sda的前100MB容量来创建这个分区，如图5-26所示。

图5-26 创建/boot分区

(4) 在创建/boot分区之后，将SDA磁盘剩余的分区以及所有剩余的物理磁盘，均配置为software RAID类型，如图5-27所示。

图5-27 配置磁盘类型

(5) 在将所有磁盘都配置成software RAID类型之后，单击Next按钮，会打开RAID Options对话框询问想要进行什么样的操作，如图5-28所示。

图5-28 设置为RAID设备

(6) 选中Create a RAID device [default=/dev/md0]单选按钮后单击OK按钮，系统弹出Make RAID Device对话框。在对话框的RAID Device

下拉列表框中，可以选择相应的RAID组在操作系统中对应的设备名。在RAID Level下拉列表框中，可以选择需要配置的RAID类型。在RAID Members列表框中，可以选择RAID组中包含的物理磁盘。用相同的方法可以做多个不同类型的RAID组，如图5-29所示。

图5-29 创建对应的Mount点

5.2 RAID卡

思考： 软件RAID有三个缺点：①占用内存空间；②占用CPU资源；③软件RAID程序无法将安装有操作系统的那个磁盘分区做成RAID模式。因为RAID程序是运行在操作系统之上的，所以在启动操作系统之前，是无法实现RAID功能的。也就是说，如果操作系统损坏了，RAID程序也就无法运行，磁盘上的数据就成了一堆无用的东西。因为RAID磁盘上的数据只有实现相应RAID算法的程序才能识别并且正确读写。如果没有相应的RAID程序，则物理磁盘上的数据仅仅是一些碎片而已，只有RAID程序才能组合这些碎片。幸好，目前大多数的RAID程序都会在磁盘上存储自己的算法信息，一旦操作系统出现了问题，或者主机硬件出现了问题，就可以将这些磁盘连接到其他机器上，再安装相同的RAID软件。RAID软件读取了存储在硬盘上固定区域的RAID信息后，便可以继续使用。

软件RAID的缺点如此之多，使人们不断地思考更多实现RAID的方法。既然软件缺点太多，那么用硬件实现如何呢？

RAID卡就是一种利用独立硬件来实现RAID功能的方法。要在硬件上实现RAID功能，必须找一个物理硬件作为载体，SCSI卡或者主板上的南桥无疑就是这个载体了。人们在SCSI卡上增加了额外的芯片用于实现RAID功能。这些芯片是专门用来执行RAID算法的，可以是ASIC这样的高成本高速度运算芯片，也可以是通用指令CPU这样的通用代码执行芯片，可以从ROM中加载代码直接执行，也可以先载入RAM后执行，从而实现RAID功能。

实现了RAID功能的板卡（SCSI卡或者IDE扩展卡）就叫做RAID卡。同样，在主板南桥芯片上也可实现RAID功能。由于南桥中的芯片不能靠CPU来完成它们的功能，所以这些芯片完全靠电路逻辑来自己运算，尽管速度很快，但是功能相对插卡式的RAID卡要弱。从某些主板的宣传广告中就可以看到，如所谓“板载”RAID芯片就是指南桥中有实现RAID功能的芯片。

这样，操作系统不需要作任何改动，除了RAID卡驱动程序之外不用安装任何额外的软件，就可以直接识别到已经过RAID处理而生成的虚拟磁盘。

对于软件RAID，至少操作系统最底层还是能感知到实际物理磁盘的，但是对于硬件RAID来说，操作系统根本无法感知底层的物理磁盘，而只能通过厂家提供的RAID卡的管理软件来查看卡上所连接的物理磁盘。而且，配置RAID卡的时候，也不能在操作系统下完成，而必须进入这个硬件来完成（或者在操作系统下通过RAID卡配置工具来设置）。一般的RAID卡都是在开机自检的时候，进入它的ROM配置程序来配置各种RAID功能。

RAID卡克服了软件RAID的缺点，使操作系统本身可以安装在RAID虚拟磁盘之上，而这是软件RAID所做不到的。

1. RAID卡的结构

带CPU的RAID卡俨然就是一个小的计算机系统，有自己的CPU、内存、ROM、总线和IO接口，只不过这个小计算机是为大计算机服务的。

图5-30为一个RAID卡的架构示意图。

图5-30 RAID卡结构示意图

SCSI RAID卡上一定要包含SCSI控制器，因为其后端连接的依然是SCSI物理磁盘。其前端连接到主机的PCI总线上，所以一定要有一个PCI总线控制器来维护PCI总线的仲裁、数据发送接收等功能。还需要有一个ROM，一般都是用Flash芯片作为ROM，其中存放着初始化RAID卡必须的代码以及实现RAID功能所需的代码。

RAM的作用，首先是作为数据缓存，提高性能；其次作为RAID卡上的CPU执行RAID运算所需要的内存空间。XOR芯片是专门用来做RAID 3、5、6等这类校验型RAID的校验数据计算用的。如果让CPU来做校验计算，需要执行代码，将耗费很多周期。而如果直接使用专用的数字电路，一进一出就立即得到结果。所以为了解脱CPU，增加了这块专门用于XOR运算的电路模块，大大增加了数据校验计算的速度。

RAID卡与SCSI卡的区别就在于RAID功能，其他没有太大区别。如果RAID卡上有多个SCSI通道，那么就称为多通道RAID卡。目前SCSI RAID卡最高有4通道的，其后端可以接入4条SCSI总线，所以最多可连接64个SCSI设备（16位总线）。

增加了RAID功能之后，SCSI控制器就成了RAID程序代码的傀儡，RAID让它干什么，它就干什么。SCSI控制器对它下面掌管的磁盘情况完全明了，它和RAID程序代码之间进行通信。RAID程序代码知道SCSI控制器掌管的磁盘情况之后，就按照ROM中所设置的选项，比如RAID类型、条带大小等，对RAID程序代码做相应的调整，操控它的傀儡SCSI控制器向主机报告“虚拟”的逻辑盘，而不是所有物理磁盘了。

提示： RAID思想中有个条带化的概念。所谓的条带化，并不是真正的像低级格式化一样将磁盘划分成条和带。这个条带化完全就是在“心中”，也就是体现在程序代码上。因为条带的位置、大小一旦设置之后，就是固定的。一个虚拟盘上的某个LBA地址块，就对应了真正物理磁盘上的一个或者多个LBA块，这些映射关系都是预先通过配置界面设定好的。而且某种RAID算法往往体现为一些复杂公式，而不是去用一张表来记录每个虚拟磁盘LBA和物理磁盘LBA的对应，这样效率会很差。因为每个IO到来之后，RAID都要查询这个表来获取对应物理磁盘的LBA，而查询速度是非常慢的，更何况面对如此大的一张表。如果用一个逻辑LBA与物理LBA之间的函数关系公式来做运算，则速度是非常快的。

正是因为映射完全通过公式来进行，所以物理磁盘上根本不用写入什么标志，以标注所谓的条带。条带的概念只是逻辑上的，物理上并不存在。所以，条带等概念只需“记忆”在RAID程序代码之中就可以了，要改变也是改变程序代码即可。唯一要向磁盘上写入的就是一些RAID信息，这样即使将这些磁盘拿下来，放到同型号的另一块RAID卡上，也能无误地认出以前做好的RAID信息。SNIA协会定义了一种DDF RAID信息标准格式，要求所有RAID卡厂家都按照这个标准来存放RAID信息，这样，所有RAID卡就都通用了。

条带化之后，RAID程序代码就操控SCSI控制器向OS层驱动程序代码提交一个虚拟化之后的所谓“虚拟盘”或者“逻辑盘”，也有人干脆称为LUN。

2. RAID卡的初始化和配置过程

所谓初始化就是说在系统加电之后，CPU执行系统总线特定地址上的第一句指令，这个地址便是主板BIOS芯片的地址。BIOS芯片中包含着让CPU执行的第一条指令，CPU将逐条执行这些指令，执行到一定阶段的时候，有一条指令会让CPU寻址总线上其他设备的ROM地址（如果有）。也就是说，系统加电之后，CPU总会执行SCSI卡这个设备上ROM中的程序代码来初始化这块卡。初始化的内容包括检测卡型号、生产商以及扫描卡上的所有SCSI总线以找出每个设备并显示在显示器上。在初始化的过程中，可以像进入主板BIOS一样，进入SCSI卡自身的BIOS中进行设置，设置内容包括查看各个连接到SCSI总线上的设备的容量、生产商、状态、SCSI ID和LUN ID等。

3. 0通道RAID卡

0通道RAID卡又称为RAID子卡，0通道的意思是说这块卡的后端没有SCSI通道。将这块子卡插入主机的PCI插槽之后，它就可以利用主板上已经集成的或者已经插在PCI上的SCSI卡，来操控它们的通道，从而实现RAID。这个0通道子卡，也是插到PCI上的一块卡，只不过它需要利用主板上为0通道子卡专门设计的逻辑电路，对外和SCSI控制器组成一块RAID卡来用，只不过这块卡在物理上被分割到了两个PCI插槽中而已。

图5-31展示了0通道RAID子卡的架构。在主板的一个特定PCI插槽上，有一个ICR逻辑电路，用来截获CPU发送的地址信号和发给CPU的中断信号。CPU发送到这里原本用来操控SCSI控制器的地址信号，现在全部被这个ICR电路重定向到了RAID子卡处，包括主板BIOS初始载入ROM，也不是载入SCSI卡的ROM了，而是载入了RAID子卡的ROM。

图5-31 0通道RAID 子卡示意图

RAID卡完全接替了SCSI卡来面对主机系统。RAID卡和SCSI控制器的通信，包括地址信息和数据信息，需要占用PCI总线，这造成了一定的性能损失。RAID子卡和SCSI卡之间的通信，不会被ICR电路重定向。

4. 无驱**RAID**卡

PhotoFast所设计的一款RAID卡可谓是比较创新。传统的RAID卡都是使用PCIX或者PCIE总线来连接到计算机上的，但是PhotoFast这款RAID卡却是使用SATA接口来连接到计算机的，也就是说，这块RAID卡将其上连接的多块物理磁盘虚拟成若干的虚拟磁盘，并将这些磁盘通过SATA接口连接到计算机，计算机就认为它自身所连接的是多块SATA物理磁盘。这样的话，这块Raid卡就不需要任何驱动程序便可被大多数操作系统使用（多数操作系统都自带SATA控制器驱动程序）。

5. **RAID On Chip**（**ROC**）技术

ROC技术是由Adaptec公司推出的一种廉价RAID技术，它利用SCSI卡上的CPU处理芯片，通过在SCSI卡的ROM中加入RAID代码而实现。

2001年，Adaptec展示了它的iROC技术，在2003年这一技术以HOSStRAID的形象推出。iROC也就是RAID on Chip，实质上就是利用SCSI控制芯片内部的RISC处理器完成一些简单的RAID类型（RAID 0、1、0+1）。由于RAID 0、1和0+1需要的运算量不大，利用SCSI控制器内部的RISC处理器也能够实现。在ROM代码的配合下，通过iROC实现的RAID 0、1或0+1具备引导能力，并且可以支持热备盘。

在入门级塔式服务器和1U高度的机架式服务器中，主板上通常会集成SCSI控制芯片，但不标配独立的RAID卡。iROC的出发点就是让这

些系统具有基本的硬件数据保护，当需要更为复杂的RAID 5时再购买独立的RAID卡。iROC的出现给低端服务器产品的数据保护方案增加了一个简易的选择。iROC或HOS tRAID的主要缺点是操作系统兼容性和性能差，由于没有专门的RAID计算处理器，因此使用这种配置的RAID会在一定程度上降低服务器系统的性能，而且它只支持RAID 0、1、0+1，只能支持几块SCSI盘做RAID，相比IDE RAID 0、1、0+1来说特性相近而成本上却高了很多，此外，HOS tRAID技术在低端还必然要面对更新、性能更好的S-ATA RAID的竞争。

6. RAID卡上的内存

RAID卡上的内存，有数据缓存和代码执行内存两种作用。

RAID卡上的CPU执行代码，当然需要RAM的参与了。如果直接从ROM中读取代码，速度会受到很大影响。所以RAID卡的RAM中有固定的地址段用于存放CPU执行的代码。而大部分空间都是用作了下文介绍的数据缓存。

缓存，也就是缓冲内存，只要在通信的双方之间能起到缓冲作用就可以了。我们知道CPU和内存之间是L2 Cache，它比内存RAM速度还要高，但是没有CPU速度快。同样，RAID控制器和磁盘通道控制器之间也要有一个缓存来适配，因为RAID控制器的处理速度远远快于通道控制器收集通道上所连接的磁盘传出的数据速度。这个缓存没有必要用L2 Cache那样高速的电路，而用RAM足矣。因为RAM的速度就足够适配二者了。

缓存RAM除了适配不同速率的芯片通信之外，还有一个作用就是缓冲数据IO。比如上层发起一个IO请求，RAID控制器可以先将这个请

求放到缓存中排队，然后一条一条地执行，或者优化这些IO，能合并的合并，能并发的并发。

7. 缓存的两种写模式

对于上层的写IO，RAID控制器有两种手段来处理，内容如下。

(1) WriteBack模式： 上层发过来的数据，RAID控制器将其保存到缓存中之后，立即通知主机IO已经完成，从而主机可以不加等待地执行下一个IO，而此时数据正在RAID卡的缓存中，而没有真正写入磁盘，起到了一个缓冲作用。RAID控制器等待空闲时，或者一条一条地写入磁盘，或者批量写入磁盘，或者对这些IO进行排队（类似磁盘上的队列技术）等一些优化算法，以便高效写入磁盘。由于写盘速度比较慢，所以这种情况下RAID控制器欺骗了主机，但是获得了高速度，这就是“把简单留给上层，把麻烦留给自己”。这样做有一个致命缺点，就是一旦意外掉电，RAID卡上缓存中的数据将全部丢失，而此时主机认为IO已经完成，这样上下层就产生了不一致，后果将非常严重。所以一些关键应用（比如数据库）都有自己的检测一致性的措施。也正因为如此，中高端的RAID卡都需要用电池来保护缓存，从而在意外掉电的情况下，电池可以持续对缓存进行供电，保证数据不丢失。再次加电的时候，RAID卡会首先将缓存中的未完成的IO写入磁盘。

(2) WriteThrough模式： 也就是写透模式，即上层的IO。只有数据切切实实被RAID控制器写入磁盘之后，才会通知主机IO完成，这样做保证了高可靠性。此时，缓存的提速作用就没有优势了，但是其缓冲作用依然有效。

除了作为写缓存之外，读缓存也是非常重要的。缓存算法是一门很复

杂的学问，有一套复杂的机制，其中一种算法叫做PreFetch，即预取，也就是对磁盘上接下来“有可能”被主机访问到的数据，在主机还没有发出读IO请求的时候，就“擅自”先读入到缓存。这个“有可能”是怎么来算的呢？

其实就是认为主机下一次IO，有很大几率会读取到这一次所读取的数据所在磁盘位置相邻位置的数据。这个假设，对于连续IO顺序读取情况非常适用，比如读取逻辑上连续存放的数据，这种应用如FTP大文件传输服务、视频点播服务等，都是读大文件的应用。而如果很多碎小文件也是被连续存放在磁盘上相邻位置的，缓存会大大提升性能，因为读取小文件需要的IOPS很高，如果没有缓存，全靠磁头寻道来完成每次IO，耗费时间是比较长的。

还有一种缓存算法，它的思想不是预取了，它是假设：主机下一次IO，可能还会读取上一次或者上几次（最近）读取过的数据。这种假设和预取完全不一样了，RAID控制器读取出一段数据到缓存之后，如果这些数据被主机的写IO更改了，控制器不会立即将它们写入磁盘保存，而是继续留在缓存中，因为它假设主机最近可能还要读取这些数据，既然假设这样，那么就没有必要写入磁盘并删除缓存，然后等主机读取的时候，再从磁盘读出来到缓存，还不如以静制动，干脆就留在缓存中，等主机“折腾”的频率不高了，再写入磁盘。

提示： 中高端的RAID卡一般具有256MB以上的RAM作为缓存。

8. RAID配置完后的初始化过程

对于校验型RAID，在RAID卡上设置完RAID参数并且应用RAID设

置之后，RAID阵列中的所有磁盘需要进行一个初始化过程，所需要的时间与磁盘数量、大小有关。磁盘越大，数量越多，需要的时间就越长。

思考： RAID卡都向磁盘上写了什么东西呢？大家可以想一下，一块刚刚出厂的新磁盘，上面有没有数据？

有。具体什么数据呢？要么全是0，要么全是1。这里所说的全0是指实际数据部分，扇区头标等一些特殊位置除外。因为磁盘上的磁性区域就有两种状态，不是N极，就是S极。那么也就是说不是0就是1，而不可能有第三种状态。那么这些0或者1，算不算数据呢？当然要算了，这些磁区不会存在一种介于0和1之间的混沌状态。如果此时用几块磁盘做了RAID 5，但磁盘上任何数据都不做改动，我们看一下此时会处于一种什么状态，比如5块磁盘，4块数据盘空间，1块校验盘空间，同一条带上，4块数据块，1块校验块，所有块上的数据都是全0，那么此时如果按照RAID 5来算，是正确的，因为 $0 \text{ XOR } 0 \text{ XOR } 0 \text{ XOR } 0 \text{ XOR } 0 = 0$ ，对。

如果一开始磁盘全是1，那么同样地 $1 \text{ XOR } 1 \text{ XOR } 1 \text{ XOR } 1 \text{ XOR } 1 = 1$ ，也对。但是如果用6块盘做RAID 5，而且初始全为1，情况就矛盾了。 $1 \text{ XOR } 1 \text{ XOR } 1 \text{ XOR } 1 \text{ XOR } 1 \text{ XOR } 1 = 0$ ，此时正确结果应该是校验块为0，但是初始磁盘全部为1，校验块的数据也为1，这就和计算结果相矛盾了。

如果初始化过程不对磁盘数据进行任何更改，直接拿来写数据，比如此时就向第二个extend上写了一块数据，将1变为0，然后控制器根据公式：新数据的校验数据 = （老数据 EOR 新数据） EOR 来校验数据。
 $(1 \text{ EOR } 0) \text{ EOR } 1 = 0$ ，新校验数据为0，所以最终数据变成了这样：1

XOR 0 XOR 1 XOR 1 XOR 1 XOR 1。我们算出它的正确数据应该等于1，而由RAID控制器算的却成了0，所以就矛盾了。

为什么会犯这个错误呢？那是因为一开始RAID控制器就没有从一个正确的数据关系开始算，校验块的校验数据一开始就与数据块不一致，导致越算越错。所以RAID控制器在做完设置，并启用之后，在初始化的过程中需要将磁盘每个扇区都写成0或者1，然后计算出正确的校验位，或者不更改数据块的数据，直接用这些已经存在的数据，重新计算所有条带的校验块数据。在这个基础上，新到来的数据才不会被以讹传讹。

思考： NetApp等产品，其RAID组做好之后不需要初始化，立即可用。甚至向已经有数据的RAID组中添加磁盘，也不会造成任何额外的IO。因为其会将所有Spare磁盘清零，也就是向磁盘发送一个Zero Unit的SCSI指令，磁盘会自动执行清零。用这些磁盘做的RAID组，不需要校验纠正，所以也不需要初始化过程，或者说初始化过程就是等待磁盘清零的过程。

关于Raid初始化过程的更详细的分析可参考本书附录1中的问与答。

9. 几款RAID卡介绍

1) Mylex AcceleRAID 352

双通道160M部门级，性能强悍，BIOS选项极为人性化，在BIOS内可以检测SCSI硬盘的出厂坏道及成长坏道，而不需借助软件。并且允许手动打开/关闭硬盘设备自身的Read cache/Write cache。还带有电池。

详细信息如下。

支持RAID级别：RAID 0、1、0+1、3、5、10、30、50、JBOD。

- 处理芯片：Intel i960RN。
- 总线类型：PCI 64b，兼容32b。
- 外置接口：Ultra 160 SCSI。
- 数据传输速率：最高160MB/s。
- 外接设备数：最多30个SCSI外设。
- 内部接口：双68针高密。
- 外部接口：双68针超高密。
- 适用的操作系统：Windows NT 4.0；Windows 2K；NetWare 4.2、5.1；SCO OpenServer 5.05、5.0.6；SCO UnixWare 7.1；DOS 6.x and above；Solaris 7（x86）；Linux 2.2 kernel distributions。
- 包括软件：Storager Manager、Storager Manager Pro和CLI（命令分界面）。
- 主要RAID特性：在线扩容、瞬时阵列可用性（后台初始化）、支持S.M.A.R.T、支持SES/SAF-TE。

图5-32和图5-33为Mylex AcceleRAID 352卡实物图。

图5-32 Mylex AcceleRAID 352卡（1）

图5-33 Mylex AcceleRAID 352卡（2）

2）LSI MegaRAID Enterprise 1600（AMI 471）

4通道160MB企业/部门级，160MB最为顶级豪华的SCSI RAID，强大的BIOS选项（LSI独有的Web BIOS）。卡上系统缓存可详细调节（除

大部分SCSI RAID可以调节的主要功能Write back（回写外），增加Read ahead（预读），Cache I/O等可调选项，满足RAID的用途需要，体现各种RAID的最高性能，带电池。

详细信息如下。

- 支持RAID级别：RAID 0、1、0+1、3、5、10、30、50、JBOD。
- 处理芯片：Intel i960RN。
- 插槽类型：PCI 64b、兼容32b。
- 总线速度：66MHz。
- 总线宽度：64b。
- 外置接口：Ultra 160 SCSI。
- 数据传输率：160MB/s。
- 最多连接设备：32。
- 内部接口：双68针高密。
- 外部接口：四68针超高密。
- 系统平台：Windows 95/98/Me/4.0/2000/XP，Linux（Red Hat、SuSE、Turbo、Caldera和FreeBSD）。

图5-34为LSI MegaRAID Enterprise 1600卡实物图。

图5-34 LSI MegaRAID Enterprise 1600卡

可以看到RAID卡使用的内存就是台式机的SDRAM内存，有些使用DDR SDRAM内存。

10. 用Rocket RAID卡做各种RAID

在一张Rocket RAID卡上，安装了8块IDE磁盘。开机之后，在启动界面按照相应提示进入RAID卡的设置界面，如图5-35所示。

图5-35 磁盘列表

可以看到，这8块硬盘有着不同的品牌、容量以及参数，但它们都是IDE接口的ATA硬盘。

1) RAID 0组的创建过程

(1) 选择RAID 0: Striping，如图5-36所示。

图5-36 选择RAID 0模式

(2) 给新RAID 0组起名为“RAID 0”，如图5-37所示。

图5-37 起名“RAID 0”

(3) 在Select Devices菜单下，选择RAID 0组所包含的磁盘，如图5-38和图5-39所示。

图5-38 选择磁盘 (1) 图5-39 选择磁盘 (2)

(4) 接下来，在Block Size菜单下可以为这个RAID 0组选择条块大小，如图5-40所示。至于Block Size参数是指整个条带的大小，还是指条带Segment的大小，要看厂家自己的定义。

图5-40 设置Block Size

(5) 选择Start Creation，确定创建RAID组，如图5-41所示。

图5-41 开始创建RAID组

(6) 创建完毕后，主界面中即显示出RAID信息，如图5-42所示。

图5-42 RAID组的信息

接下来我们继续用以上方法创建其他类型的RAID组。

2) RAID 1组的创建过程

图5-43所示是RAID 1组的创建过程，可以发现Start Creation中有一个Duplication选项，这个选项的作用是将源盘数据复制到镜像盘，而不破坏源盘数据。如果选择了Create Only，则会破坏源盘的数据，重新创建干净的RAID 1组。

图5-43 创建RAID 1组

3) 创建一个3块盘组成的RAID 5组

提示： 在Start Creation菜单中有两个选项，一个为Zero Build，另一个为No Build，如图5-44所示。Zero Build指将所有数据作废，从零开始生成数据的校验值。No Build指不计算数据校验值，如果用户能保证RAID 5组中的磁盘原来是处于一致性状态的，则可以用这个选项来节约时间，否则不要选择这个选项。

图5-44 两个选项

如果选择No Build选项，则会显示警告信息，如图5-45所示。

图5-45 警告信息

按Y键即可完成RAID 5组的创建。

至此，我们创建了RAID 0、RAID 1和RAID 5三个RAID组，如图5-46所示。

图5-46 三个RAID组的信息

4) 删除RAID组

如果对创建的RAID组不满意，可以删除重建，具体操作如图5-47和图5-48所示。

图5-47 删除RAID组 图5-48 确认信息

5) 添加全局热备磁盘

此外，还可以添加全局热备磁盘。切换到Add/Remove Spare菜单，如图5-49所示。

图5-49 添加全局热备盘

由于当前系统中只有一块空闲磁盘，所以我们就将这块磁盘作为全局热备磁盘，操作如图5-50和图5-51所示。如果任何RAID组中有磁盘损坏的话，RAID卡将利用这块热备磁盘来顶替损坏的磁盘，将数据重新同步到这块磁盘上。

图5-50 确认信息 图5-51 磁盘状态

6) 设置启动标志

由于系统要从安装有操作系统的磁盘上启动，所以必须让RAID卡知道哪个逻辑磁盘是启动磁盘。具体设置如图5-52～图5-54所示。

图5-52 设置启动盘（1）

图5-53 设置启动盘（2）

图5-54 设置启动盘（3）

在将RAID 1组形成的逻辑磁盘作为启动磁盘后，可以看见右边的“BOOT”标志。

7) 设置访问各个磁盘的模式参数

在Device Mode菜单下，可以设置访问各个磁盘的模式参数，如图5-55和图5-56所示。

图5-55 设置磁盘参数（1）

图5-56 设置磁盘参数（2）

8) 查看所有设备

在View菜单下，可以查看所有设备、所有RAID组和所有逻辑磁盘（由于这块卡不具有在RAID组中再次划分逻辑磁盘的功能，所以每个逻辑组只能作为一个逻辑磁盘），如图5-57和图5-58所示。

图5-57 RAID组状态（1）

图5-58 RAID组状态（2）

5.3 磁盘阵列

RAID卡的出现着实让存储领域变得红火起来，几乎每台服务器都标配RAID卡或者集成的RAID芯片。一直到现在，虽然磁盘阵列技术高度发展，各种盘阵产品层出不穷，但RAID卡依然是服务器不可缺少的一个部件。

然而，RAID卡所能接入的通道毕竟有限，因此人们迫切希望创造一种可以接入众多磁盘、可以实现RAID功能并且可以作为集中存储的大规模独立设备。最终，磁盘阵列在这种需求中诞生了。

磁盘阵列的出现是存储领域的一个里程碑。关于磁盘阵列的描述，我们将在本书第6章中详细介绍。

在7种RAID形式的基础上，还可以进行扩展，以实现更高级的RAID。由于RAID 0无疑是所有RAID系统中最快的，所以将其他RAID形式与RAID 0杂交，将会生成更多新奇的品种。将RAID 0与RAID 1结合，生成了RAID 10；将RAID 3与RAID 0结合，生成了RAID 30；将RAID 5与RAID 0结合，生成了RAID 50。

5.3.1 RAID 50

图5-59是一个RAID 50的模型，RAID 30与其类似。控制器接收到主机发来的数据之后，按照RAID 0的映射关系将数据分块，一部分存放于左边的RAID 5系统，另一部分存放在右边的RAID 5系统。左边的RAID 5系统再次按照RAID 5的映射关系将这一部分数据存放于5块磁盘中的若干块，另一边也进行相同的过程。

图5-59 RAID 50模型

实际中，控制器不可能物理地进行两次运算和写IO，这样效率很低。控制器可以将RAID 0和RAID 5的映射关系方程组合成一个函数关系方程，这样直接代入逻辑盘的LBA，便可得出整个RAID 50系统中所有物理磁盘将要写入或者读取的相应LBA地址，然后统一向磁盘发送指令。左边的RAID 5系统和右边的RAID 5系统分别允许损坏一块磁盘而不影响数据。但是如果在任何一边的RAID系统同时或者先后损坏了两块或者更多的盘，则整个系统的数据将无法使用。

5.3.2 RAID 10和RAID 01

RAID 10和RAID 01看起来差不多，但是本质上有一定区别。图5-60是一个RAID 10的模型。

图5-60 RAID 10模型

如果某时刻，左边的RAID 1系统中有一块磁盘损坏，此时允许再次损坏的磁盘就剩下两块，也就是右边的RAID 1系统中还可以再损坏任意一块磁盘，而整体数据仍然是可用的。我们暂且说这个系统的冗余度变成了2。

图5-61是一个RAID 01的模型。

图5-61 RAID 01模型

如果某时刻，左边的RAID 0系统中有一块磁盘损坏，此时左边的RAID 0系统便没有丝毫作用了。所有的IO均转向右边的RAID 0系统。而此时，仅仅允许左边剩余的那块磁盘损坏。如果右边任何一块磁盘损

坏，则整体数据将不可用。所以这个系统的冗余度变成了1，即只允许损坏特定的一块磁盘（左边RAID 0系统剩余的磁盘）。

综上所述，RAID 10系统要比RAID 01系统冗余度高，安全性高。

5.4 虚拟磁盘

话说张真人送走了七星大侠之后，面对江湖上的浮躁，有苦难言。这江湖还能出一个像七星这样的豪侠吗？难啊！七星北斗阵，多么完美的一个阵式！七星老前辈用尽毕生心血，创立了7种阵式，将单个磁盘组成盘阵，提高整体性能！可是很少有人能体会到这个阵式的精髓，包括创建他的七星，都不一定。张真人自从七星走后，一直处于深度悲痛之中，悔恨当初为什么没有抽时间向七星拜师学艺！如今只能守着一本老侠留下来的《七星北斗阵式》天天仔细研读，以求找到什么灵感，来继续发扬老侠的这门绝技。

.....

就这样过去了20年。张真人已经由年轻小伙变成了稳重善思的中年人。他凭借优秀的武艺和才华，来到武当山创立了道观，并收下了7位徒弟，以纪念七星北斗之豪情！张真人每晚休息之前，都要对着七星北斗拜三拜。20多年过去了，北斗的光芒依然是那么耀眼，依然看着世间纷争，昼夜交替。

这20年是科技飞速发展的20年。铁匠们的技艺提高很快，新技术不断被创造出来。大容量、高速度的磁盘在地摊卖10文钱一斤。

某天张老道下山溜达，发现地摊上的磁盘品质还不错，比20年前的货强太多了，顺手就买了50斤回去。点了点，足足50块。他让他的7位徒弟，分别按照七星阵摆上各种阵形，来捣鼓这50块硬盘。7位徒弟早就对七星阵烂熟于心，把这50块磁盘捣鼓得非常顺。张老道频频点头，心里想着：“嗯，真应了那句话啊。长江后浪推前浪，一代新人换旧

人！”摆弄了一阵之后，徒弟们都累了。这次格外地累，不禁都坐在地上休息。老道把眼一瞪，“嗯忒！！！年轻人，不好好练功！不准偷懒！”徒弟们上前道：“师父，不是我们偷懒，这次您买的磁盘和以前的不一样。我们在出招的时候，就是在‘化龙’这一招的时候特别吃力。这条龙太大，不好操控。”老道一看，果然，这50块磁盘每块足有1TB大，50块就是50TB。“嚯嚯，20年前一块磁盘最多也就是50MB，没想到啊！”

这天晚上，老道用完粗茶淡饭之后，遥望北斗，心想：七星老侠在天上不知道看见此情此景，会给我什么启示呢？20年前，用此阵式生成的虚拟磁盘，大小也不过几GB，而如今已经达到了TB级别，也难怪我那些徒儿们会吃不消。怎么办呢？需要把这以TB论的虚拟磁盘再次划分开来，划分成多条“小龙”，这样就可以灵活操控了。而且针对目前的磁盘超大的容量，完全可以在一个阵中同时应用多种阵式。比如让我7位徒弟，其中3人摆出RAID 0阵式，另外4人同时摆出RAID 5阵式，共同出招。对每个阵式生成虚拟“龙盘”，把它划分成众多小的“龙盘”，这样对外不但我们的威力没有减少，而且可以灵活运用，让敌人不知道我们到底有几个人。

张老道决定将大龙盘划分成小龙盘，这事十分好办，只需体现在“心中”就可以了。只要你心中有数，那些物理磁盘的哪部分区域属于哪个小龙盘，就完全可以对外通告了。老道称这种技术为逻辑盘技术。

5.4.1 RAID组的再划分

实际中，比如用5块100GB的磁盘做了一个RAID 5，那么实际数据空间可以到400GB，剩余100GB空间是校验空间。如果将这400GB虚拟成一块盘，不够灵活。且如果OS不需要这么大的磁盘，就没法办了。

所以要再次划分这400GB的空间，比如划分成4块100GB的逻辑磁盘。而这逻辑盘虽然也是100GB，但是并不同于物理盘，向逻辑盘写一个数据会被RAID计算，而有可能写向多块物理盘，这样就提升了性能，同时也得到了保护。纵使RAID组中坏掉一块盘，操作系统也不会感知到，它看到的仍然是100GB的磁盘。

5.4.2 同一通道存在多种类型的RAID组

不仅如此，老道还想到了在一个阵式中同时使用多种阵法的方式。

实际中，假设总线上连接有8块100GB的磁盘，我们可以利用其中的5块磁盘来做一个RAID 5，而后再利用剩余的3块磁盘来做一个RAID 0，这样，RAID 5的可用数据空间为400GB，校验空间为100GB，RAID 0的可用数据空间为300GB。而后，RAID 5和RAID 0各自的可用空间，又可以根据上层OS的需求，再次划分为更小的逻辑磁盘。这样就将七星北斗阵灵活地运用了起来，经过实践的检验，这种应用方法得到了巨大的推广和成功。

张老道给划分逻辑盘的方法取名为巧化神龙，将同一个阵中同时使用多种阵式的方法叫做神龙七变。

5.4.3 操作系统如何看待逻辑磁盘

目前各种RAID卡都可以划分逻辑盘，逻辑盘大小任意设置。每个逻辑盘对于OS来说都认成一块单独的物理磁盘。这里不要和分区搞混，分区是OS在一块物理磁盘上做的再次划分。而RAID卡提供给OS的，任何时候，都是一块或者几块逻辑盘，也就是OS认成的物理磁盘。而OS在这个磁盘上，还可以进行分区、格式化等操作。

5.4.4 RAID控制器如何管理逻辑磁盘

下面说一下RAID卡对逻辑磁盘进行再次划分的具体细节。既然要划

图5-62 DDF 布局图

分，就要心中有数，比如某块磁盘的某个区域，划分给哪个逻辑盘用，对应逻辑盘的LBA地址是多少，这块磁盘的RAID类型是什么等。而这些东西不像RAID映射那样根据几个简单的参数就能确定，而且对应关系是可以随时变化的，比如扩大和缩小、移动等。所以有必要在每块磁盘上保留一个区域，专门记录这种逻辑盘划分信息、RAID类型以及组内的其他磁盘信息等，这些信息统称为RAID信息。不同厂家、不同品牌的产品实现起来不一样，SNIA委员会为了统一RAID信息的格式，专门定义了一种叫做DDF的标准，如图5-62所示。

图5-63所示的是微软和Veritas公司合作开发的软RAID在磁盘最末1MB空间创建的数据结构。有了这个记录，RAID模块只要读取同一个RAID子系统中每块盘上的这个记录，就能够了解RAID信息。即使将这些磁盘打乱顺序，或者拿到其他支持这个标准的控制器上，也照样能够认到所划分好的逻辑盘等所有需要的信息。

图5-63 Windows系统中的动态磁盘信息

RAID卡可以针对总线上的某几块磁盘做一种RAID类型，然后针对另外的几块磁盘做另一种RAID类型。一种RAID类型中包含的磁盘共同组成一个RAID Group，简称RG。逻辑盘就是从这个RG中划分出来的，原则上逻辑盘不能跨RG来划分，就是说不能让一个逻辑盘的一部分处于一个RG，另一部分处于另一个RG。因为RG的RAID类型不一样，其性能也就不一样，如果同一块逻辑盘中出现两种性能，对上层应用来说

不是件好事，比如速度可能会忽快忽慢等。

张真人推出了这两门绝技之后，在江湖上引起了轩然大波。大家争相修炼，并取得了良好的效果。一时间，江湖上几乎人人都练了张真人这两门功夫。而且各大门派已经将七星北斗阵以及张真人的功夫作为各派弟子必须掌握的基本功。

近水楼台先得月。武当七子当然已经把功夫练到了炉火纯青的地步。老道非常欣慰。他相信七星侠在天之灵倘若看到了这阵式被拓展，一定也会感到欣慰的。

5.5 卷管理层

老道创立这两门功夫的兴奋，很快就被一个不大不小的问题给吹得烟消云散。这个问题就是一旦逻辑盘划分好之后就无法改变，要改变也行，上面的数据就得全部抹掉，这是让人无法容忍的。比如已经做好了一个100GB的逻辑盘，但是用了两年以后，发现数据越来越多，已经盛不下了。但又不能放到别的磁盘，因为受上层文件系统的限制，一个文件不可能跨越多个分区来存放，更别提跨越多个磁盘了。如果有一个文件已经超过了100GB，那么谁也无回天，只能重新划分逻辑盘。数据怎么办？这问题遇不到则已，遇到了就是死路一条。江湖上已经有不少生意人因为这个问题而倾家荡产，他们无奈之余，准备联合起来到武当恳求张老道想一个办法，以克服这个难关，好让他们东山再起。张老道对他们的遭遇深感同情，同时也责怪自己当初疏忽了这个问题。于是他当众许下承诺：3个月之后，来武当取解决办法。

5.5.1 有了逻辑盘就万事大吉了么？

1. 踏破铁鞋无觅处——寻找更加灵活的磁盘卷管理方式

其实张真人许下3个月的时间，他自己也毫无把握。但是为了平息众怒，也只能冒险赌一次了！送走众人之后，张老道就开始天天思考解决这个问题的办法。他想：到底怎么样才能让使用者运用自如呢？如果一开始就给它划分一个100GB的逻辑盘，如果数据盛不下了，此时把他磁盘上未使用的空间挪一部分到这个逻辑盘，岂不是就可以了么？

可以是，但从RAID卡设置里增加或减少逻辑盘容量很费功夫。在RAID卡里增加这种代码，修炼成本很高，而且即使实现了，主

机也不能立即感应到容量变化。即使感应到了，也不能立即变更。对于Windows系统来说，必须将其创建为新的分区。想要合并到现有分区，必须用第三方分区表调整工具在不启动操作系统的情况下来修改分区表才行。再者，其上的文件系统不一定会跟着扩大，NTFS这种文件系统不能动态张缩，也必须在不启动操作系统的条件下用第三方工具调整。这种方法对一些要求不间断服务的应用服务器并不适用。

老道想到这里，觉得至少已经找到了一种解决办法，虽然不是很方便，需要重启主机，之后再在RAID卡中更改配置。更改完毕后，可能还要重启一次，然后进入系统，系统才能认出新容量的磁盘。而OS就算正确认出了新增的磁盘容量，由于分区表没有改变，新增容量不属于任何一个分区，还是不能被使用，所以还需要手动修改分区表。太复杂、太麻烦了，能否找一种方便快捷的方法呢？

2. 得来全不费工夫——来源于现实的刺激

话说冬至这天，天上飘着雪花。武当山张灯结彩，喜气洋洋！这天是张真人的70大寿！江湖各大门派及各路英雄纷纷前来拜寿。武当上下忙得是不亦乐乎！就说包饺子吧，一会儿面不够了去和面，一会儿水不够了去挑水。张真人是往来作揖，笑迎来宾。厨房则加紧和面，由于厨房空间太小，所以和好的面被运往各个分理点处，那里有小道士负责擀皮、包饺子。张真人看着眼前这小老道跑来跑去的多少回了，就纳闷了，所以跟着去看看怎么回事。一看才知道，弄了半天是往各处运面团呢！觉得挺好笑的，也没当回事。等大家都差不多到齐了，共同给老道祝了寿，然后就上饺子了。张老道看着碗里一个个的饺子，再想想刚才那面团的事，心里突然一动！于是当众宣布，一个月前自己承诺的约定过不了几天就会实现了！众豪杰都鼓起掌来！

提示： 张老道到底想到了什么呢？原来，他想起了小道包饺子和面的情形。厨房和了一大团面，下面随用随取。不够了，割一块揉进去就行了，或者掰下一块来放着下次用。这不正解决了一个月前大家所头疼的问题吗？RAID控制器和好了几团面（逻辑盘），放那由自己看着用，哪不够了就掰块补上。必须实现这样一种像掰面团一样灵活的管理层，才能最终解决使用中出现的問題。是啊，说得简单，可是具体要做却不是那么容易的。

当天晚上，老道睡觉的时候就一个劲地想，在RAID控制器上掰面，以前也分析过了，不合适，那么在哪里掰呢？RAID控制器给你和了几斤面，你就得收着，不要也不行。但是面收着了，你可以自己掰呀，是啊，自己掰。那么就是说，RAID控制器提交给OS的逻辑磁盘。应该可以掰开，或者揉搓到一块儿去，可以想怎么揉搓就怎么揉搓。这功能如果能通过在操作系统上运行一层软件来实现的话，不但灵活，而且管理方便！想到这，老道心里有了底。

第二天，老道就让徒弟们按照他写的口诀来实现他这个想法，大获成功！RAID控制器是硬件底层实现RAID，实现逻辑盘，所以操作起来不灵活。如果在OS层再把RAID控制器提交上来的逻辑盘（OS会认成不折不扣的物理磁盘）加以组织、再分配，就会非常灵活。因为OS层上运行的都是软件，完全靠CPU来执行，而不用考虑太多的细节。张老道立即将这种新的掌法公布天下，称作神仙驾龙！

5.5.2 深入卷管理层

实际中，有很多基于这种思想的产品，这些产品都有一个通用的名称，叫做卷管理器（Volume Manager, VM）。比如微软在Win2000中

引入的动态磁盘，就是和Veritas公司合作开发的一种VM，称为LDM（逻辑磁盘管理）。Veritas自己的产品Veritas Volume Manager（VxVM）和广泛用于Linux、AIX、HPUX系统的LVM（Logical Volume Manager），以及用于Sun Solaris系统的Disk Suite，都是基于这种在OS层面，将OS识别到的物理磁盘（可以是真正的物理磁盘，也可以是经过RAID卡虚拟化的逻辑磁盘）进行组合，并再分配的软件。它们的实现方法大同小异，只不过细节方面有些差异罢了。

这里需要重点讲一下LVM，因为它的应用非常普遍。LVM开始是在Linux系统中的一种实现，后来被广泛应用到了AIX和HPUX等系统上。

- **PV**：LVM将操作系统识别到的物理磁盘（或者RAID控制器提交的逻辑磁盘）改了个叫法，叫做Physical Volume，即物理卷（一块面团）。
- **VG**：多个PV可以被逻辑地放到一个VG中，也就是Volume Group卷组。VG是一个虚拟的大存储空间，逻辑上是连续的，尽管它可以由多块PV组成，但是VG会将所有的PV首尾相连，组成一个逻辑上连续编址的大存储池，这就是VG。
- **PP**：也就是Physical Partition（物理区块）。它是在逻辑上再将一个VG分割成连续的小块（把一大盆面掰成大小相等的无数块小面团块）。注意，是逻辑上的分割，而不是物理上的分割，也就是说LVM会记录PP的大小（由几个扇区组成）和PP序号的偏移。这样就相当于在VG这个大池中顺序切割，如果设定一个PP大小为4MB，那么这个PP就会包含8192个实际物理磁盘上的扇区。如果PV是实际的一块物理磁盘，那么这些扇区就是连续

的。如果PV本身是已经经过RAID控制器虚拟化而成的一个LUN，那么这些扇区很有可能位于若干条带中，也就是说这8192个扇区物理上不一定连续。

- **LP**: PP可以再次组成LP，即Logical Partition（逻辑区块）。逻辑区块是比较难理解的，一个LP可以对应一个PP，也可以对应多个PP。前者对应前后没什么区别。后者又分两种情况：一种为多个PP组成一个大LP，像RAID 0一样；另一种是一个LP对应几份PP，这几份PP每一份内容都一样，类似于RAID 1，多个PP内容互为镜像，然后用一个LP来代表它们，往这个LP写数据，也就同时写入了这个LP对应的几份PP中。
- **LV**: 若干LP再经过连续组合组成LV（Logical VoLUNme，逻辑卷），也就是LVM所提供的最终可用来存储数据的单位。生成的逻辑卷，在主机看来还是和普通磁盘一样，可以对其进行分区、格式化等。

思考： 有人问了，一堆面团揉来揉去，最终又变成一堆面团了，你这是揉面还是做存储呢？

确实，面团最终还是面团。但是此面团非彼面团。最终形成的这个LV，它的大小可以随时变更，也不用重启OS，你想给扩多大就扩多大，前提是面盆里面还有被掰开备用的PP。而且，只要盆里面有PP，你就可以再创建一个LV，也就是再和一团面，LV数量足够用的。如果不增加卷管理这个功能，那么RAID卡提交上来多少磁盘，容量多大就是多大，不能在OS层想改就改、为所欲为。而卷管理就提供了这个为所欲为的机会，让你随便和面团。

LVM看起来很复杂，其实操作起来很简单。创建PV，将PV加入VG，在VG中再创建LV，然后格式化这个LV，就可以当成一块普通硬

盘使用了。容量不够了，还可以随便扩展，岂不快哉？LVM一个最大的好处就是生成的LV可以跨越RAID卡提交给OS的物理磁盘（逻辑盘）。这是理所当然的，因为LVM将所有物理盘都搅和到一个大面盆中了，当然就可以跨越物理盘了。

5.5.3 Linux下配置LVM实例

下面以RedHat Enterprise Linux Server 4 Update 5操作系统为例，给大家示例一下LVM的配置过程。

（1）在操作系统安装过程中，选择手动配置磁盘管理，如图5-64所示。

图5-64 选择手动管理

（2）可以看到，这台机器共有8块物理磁盘，每块的容量为1GB，如图5-65所示。

图5-65 磁盘列表

（3）首先，需要定义一个/boot分区，这个分区是用来启动基本操作系统内核的，所以这块空间不能参与LVM。我们选择从第一块硬盘（sda）划分出20个磁道的空间用来作为/boot分区。这块空间也就成了sda1设备，如图5-66所示。

图5-66 创建/boot分区

（4）接下来，对于sda2、sdb、sdc、sdd、sde、sdf、sdg、sdh所有这些剩余的磁盘或者分区，就可以将它们配置成LVM的PV（物理卷）。选中每个磁盘或者分区，单击Edit按钮。在File System Type下拉

列表框中，选择physical volume（LVM）选项，表示将这个硬盘或者分区配置成LVM的PV。PV可以任意设定大小，只要编辑End Cylinder文本框中的值即可。剩余空间可以继续作为PV再次分配。对每个磁盘都进行上述操作，如图5-67所示。

图5-67 设置磁盘类型为LVM管理状态

（5）操作完成后，信息栏中显示所有磁盘和sda2分区都已被配置成为PV，如图5-68所示。

图5-68 磁盘状态

（6）单击LVM按钮会出现如图5-69所示的对话框。这一步就是创建VG（Volume Group，卷组）的过程。可以将PV进行任意组合，组合后的PV就形成了VG。

图5-69 创建卷组

（7）这里我们做一个名为“VolGroup00”的卷组，其包含sda2和sdb1两个PV。在Physical Extent下拉列表框中，可以选择这个卷组对应的磁盘空间的最小分配单位（在AIX的LVM中，这个最小单位称为Physical Partition，即PP）。然后单击下方的Add按钮，从这个大的卷组空间中再次划分逻辑卷，即LV。下面创建一个大小为1000MB的逻辑卷LogVol00，并且用ext3文件系统将这个卷格式化，并挂载到/home目录下，如图5-70所示。

图5-70 创建LV并挂载

（8）然后将VolGroup00卷组中剩余的空间，全部分配给一个新的LV，即LogVol01，用ext3文件系统格式化，并挂载到/tmp目录下，如图

5-71所示。

图5-71 创建LV并挂载

(9) 将剩余的sdc1、sdd1、sde1、sdf1、sdg1、sdh1这几块PV全部分配给一个新的卷组VolGroup01，并且在卷组中创建一个逻辑卷LogVol00，大小为整个卷组的大小，用ext3文件系统格式化，并挂载到目录下，如图5-72所示。

图5-72 创建LV并挂载

(10) 配置完成后的状态如图5-73所示。

图5-73 配置完成后的状态

5.5.4 卷管理软件的实现

说到这里，别以为LVM就只会像疯子一样，拿来面团就揉到一起，掰来掰去，什么都不管，那样岂不真成了马大哈了。它是需要在心里暗自记录的，比如某块物理盘的名称和容量。表面上是和其他物理盘融合到一起，但还是要记住谁是谁，从哪里到哪里属于这块盘，从哪里到哪里属于那块盘，地址多少，等等。

这些信息记录在磁盘上的某个区域，LVM中这个区域叫做VGDA。LVM就是通过读取每块物理磁盘上的这个区域来获取LVM的配置信息，比如PP大小、初始偏移、PV的数量和信息、排列顺序及映射关系等。LVM初始化的时候会读取这些信息，然后在缓存中生成对应的映射公式，从而完成LV的挂载。挂载之后，就可以接受IO了。比如上层访问某个LV的LBA 0xFF地址，那么LVM就需要通过缓存中的映射关系

判断这个地址对应到实际物理磁盘是哪个或哪几个实际地址。假设这个地址实际对应了磁盘a的LBA 0xAA地址，那么就会通过磁盘控制器驱动直接给这个地址发数据，而这个地址被RAID控制器收到后，可能还要做一次转换。因为OS层的“物理磁盘”可能对应真正的存储总线上的多块物理磁盘，这个映射就需要RAID控制器来做了，原理都是一样的。

卷管理软件对待由RAID卡提交的逻辑盘（OS识别成物理磁盘）和切切实实的物理盘的方法是一模一样的。也就是说，不管最底层到底是单物理盘，还是由RAID控制器提交的逻辑物理盘，只要OS认成它是一块物理磁盘，那么卷管理器就可以对它进行卷管理。只不过对于RAID提交的逻辑盘，最终还是要通过RAID控制器来和最底层的物理磁盘打交道。

Linux下的LVM甚至可以对物理磁盘上的一个分区进行卷管理，将这个分区做成一个PV。

卷管理软件就是运行在OS操作系统磁盘控制器驱动程序之上的一层软件程序，它的作用就是实现RAID卡硬件管理磁盘空间所实现不了的灵活功能，比如随时扩容。

为什么卷管理软件就可以随时在线扩容，灵活性这么强呢？首先我们要熟悉一个知识，也就是OS会自带一个卷管理软件层，这个卷管理软件非常简单，它只能管理单个磁盘，而不能将它们组合虚拟成卷，不具有高级卷管理软件的一些灵活功能。OS自带的一些简单VM（卷管理）软件，只会调用总线驱动（一种监视IO总线Plug And Play，即PNP，即插即用），发现硬件之后再挂接对应这个硬件的驱动，然后查询出这个硬件的信息，其中就包括容量，所以我们才会在磁盘管理器中看到一块块的磁盘设备。即从底层向上依次是物理磁盘、磁盘控制器、

IO总线、总线驱动、磁盘控制器驱动、卷管理软件程序、OS磁盘管理器中看见的磁盘设备。

而高级卷管理软件是将原本OS自带的简陋的卷管理功能进行了扩展，比如可以对多个磁盘进行组合、再分等。不管是OS单一VM还是高级VM，磁盘在VM这一层处理之后，应该称为卷比较恰当，就算卷只由一块磁盘抽象而成，也不应该再称作磁盘了。因为磁盘这个概念只有对磁盘控制器来说才有意义。

磁盘控制器看待磁盘，真的就是由盘片和磁头组成。而卷管理软件看待磁盘，会认为它是一个线性存储的大仓库，而不管这个仓库用的是什么存储方式，仓库每个房间都有一个地址（LBA逻辑块地址），VM必须知道这些地址一共有多少。它让库管员（磁盘控制器驱动软件）从某一段地址（LBA地址段）存取某些货物（数据），那么库管员就得立即操控他的机器（磁盘控制器）来到各个房间存取货物（数据）。这就是VM的作用。

在底层磁盘扩容之后，磁盘控制器驱动程序会和VM打个招呼：我已经增大了多少容量了，你看着办吧。卷说：“好，你不用管了，专心在那干活吧，我告诉你读写哪个LBA地址的数据你就照我的话办。”这样之后，VM就会直接将等待扩容的卷的容量立即扩大，放入池中备用，对上层应用没有丝毫影响。所以VM可以屏蔽底层的变化。

至于扩容和收缩逻辑卷，对VM来说是小事一桩。但是对于其上的文件系统来说，处理起来就复杂了。所以扩大和收缩卷，需要其上的文件系统来配合，才能不影响应用系统。

5.5.5 低级VM和高级VM

1. MBR和VGDA

分区管理可以看作是一种最简单的卷管理方式，它比LVM等要低级。分区就是将一块磁盘抽象成一个仓库，然后将这个仓库划分成具体的一库区、二库区等。因为一个仓库太大的话，对用户来说很不方便。比如一块100GB的磁盘，如果只分一个区，就显得很不便于管理。有两种方法解决这个问题：

（1）可以用低级VM管理软件，比如Windows自带的磁盘管理器，对这个磁盘进行分区；

（2）用高级VM管理软件，将这个盘做成卷，然后灵活地进行划分逻辑卷。

这两种方法可以达到将一个仓库逻辑划分成多个仓库的效果。所不同的是分区管理这种低级卷管理方式，只能针对单个磁盘进行再划分，而不能将磁盘合并再划分。

思考：对于低级VM的分区管理来说，必须有一个东西来记录分区信息，如第一仓库区是整个仓库的哪些房间，从第几个房间开始到第几个结束是第二仓库区等这些信息。这样，每次OS启动的时候，VM通过读取这些信息就可以判断这个仓库一共有几个逻辑区域，从而在“我的电脑”中显示出逻辑磁盘列表。那么怎么保存这个分区信息呢？

毫无疑问，它不能保存在内存里，更不能保存在CPU里，它只能保存在磁盘上。分区信息被保存在分区表中，分区表位于磁盘0磁道0磁头的0号扇区上，也就是LBA1这个地址的扇区上。这个扇区又叫做MBR，即主引导记录。MBR扇区不仅仅保存分区表，它还保存了BIOS

跳转时所需要执行的第一句指令代码，所以才叫做主引导记录。

BIOS代码都是固定的，它每次必定要执行LBA1扇区上的代码。如果修改BIOS，让它执行LBA100扇区的代码，也可以，完全可以。但是现在的BIOS都是执行LBA1处的代码，没人去改变。而新出的规范EFI将要取代BIOS，并且在安腾机上已经使用了，一些苹果笔记本也开始使用EFI作为BIOS的替代。在EFI中可以灵活定制这些选项，比如从哪里启动，不仅可以选择设备，还可以选择设备上的具体地址。

MBR中除了包含启动指令代码，还包含分区表。通常启动时，程序都会跳转到活动分区去读取代码做OS的启动，所以必须有一个活动分区。这在分区工具中可以设置。

高级卷管理软件在划分了逻辑卷之后，一定要记录逻辑卷是怎么划分的，比如LVM就需要记录PV的数量和信息、PP的大小、起始位置及LV的数量和信息等。这些信息都要保存在磁盘上，所以也要有一个数据结构来存储。这个数据结构，LVM使用VGDA（Volume Group Descriptor Area）。每次启动系统，VM就是通过读取这些数据来判断目前的卷情况并挂载LV的。VGDA的大致结构示意图如图5-74所示。

图5-74 VGDA示意图

不管是MBR中的分区表，还是VGDA9数据结构，一旦这些信息丢失，逻辑卷信息就会丢失，整个系统的数据就不能被访问。

低级VM在给磁盘分区的时候，会更新MBR中的分区表；高级VM做逻辑卷的时候，同样也会更新VGDA中的数据。其实高级VM初始化一组新磁盘的时候，并没有抛弃MBR。因为它们除了写入VGDA信息之

外，也要更新MBR扇区中的分区表，将用于启动基本操作系统的代码单独存放到一个分区中，并标明分区类型为bootable类型，证明这个分区是用于在卷管理模块还没有加载之前启动操作系统的。并将磁盘所有剩余容量划分到一个分区中，并标明这个分区的类型，如AIX类型。

在安装Linux的时候，必须单独划分一个/boot分区，这个分区就是用于启动基本操作系统用的，100MB大小足矣。启动操作系统所必需的代码都放在这个分区。同样AIX系统也要保留一个分区用来启动最基本的操作系统代码。这也是AIX在进行了Mirrorvg镜像操作之后，需要执行BOSboot命令来写入boot分区的内容的原因，因为boot分区没有参与VM管理。这个启动分区是不能做到VM中的，因为VM代码不是在BIOS将控制交给OS的时候一开始就执行的。

总之，高级VM没有抛弃MBR分区的解决方案，而是在MBR基础上，又增加了类似VGDA这种更加灵活的数据结构来动态管理磁盘。

2. RAID功能

高级VM软件一般均带有软RAID功能，可以实现逻辑卷之间的镜像。更有甚者，有些VM甚至实现了类似RAID 0的条带化。在卷的级别条带化，达到在物理盘级别条带化同样的目的。但是如果磁盘已经被硬件RAID控制器条带化过了，并且这些LUN是在一个RAID Group中，那么VM再来条带化一下子不但没有必要，而且可能二次条带化会将效果抵消。

Windows的动态磁盘VM还可以以纯软件方式实现RAID 5，所有计算都靠CPU，所以也就注定了它比硬件RAID更灵活，但在高系统负载的情况下，它相比硬件RAID来说速度和性能稍差。

5.5.6 VxVM卷管理软件配置简介

VxVM是Veritas公司开发的一个高级卷管理软件，支持RAID 0、RAID 1、RAID 01和RAID 5四种软RAID模式，支持动态扩大和缩小卷容量。

下面的例子是在一个UNIX系统中对4块磁盘做卷管理的案例。所有命令均在UNIX的Shell下执行。

1. 创建磁盘组

磁盘组就是将所有磁盘作为一个大的资源池，卷将在这个池中产生。

首先，初始化硬盘。

- `vxdisksetup -i disk1`
- `vxdisksetup -i disk2`
- `vxdisksetup -i disk3`
- `vxdisksetup -i disk4`

然后创建一个名为“DataDG”的磁盘组，该磁盘组包含了disk1、disk2、disk3和disk4四个磁盘。

- `vx dg init DataDG disk1 disk2 disk3 disk4`

除了这种方法，用户还可以用以下方法来创建磁盘组。

- `vx dg init DataDG DataDG01=disk1`（创建一个只包含disk1的磁盘组）

- `vx dg -g DataDG adddisk DataDG02=disk2`（将disk2加入到该磁盘组）
- `vx dg -g DataDG adddisk DataDG03=disk3`（将disk3加入到该磁盘组）
- `vx dg -g DataDG adddisk DataDG03=disk4`（将disk4加入到该磁盘组）

如果用户在所需磁盘空间不足，需要扩容的时候，利用添加磁盘到磁盘组的方法，就可以在不破坏现有环境的情况下扩大系统的容量。

2. 创建卷

创建卷必须指明在哪个磁盘组下面创建，最常用的方法如下：

- `vxassist -g DataDG make DataVolA 5g`

该命令将在DataDG磁盘组上创建名为“DataVolA”的卷，卷的大小是5GB。

如果用户希望该卷只创建在disk1和disk2上面，不占用disk3和disk4的空间，那么可以执行下列命令。

- `vxassist -g DataDG make DataVolA 5g disk1 disk2`

创建一个5GB大小的条带卷（RAID 0）。

- `vxassist -g DataDG make DataVolB 5g layout=stripe`

这样就在DataDG磁盘组上面建立了一个名为“DataVolB”的5GB大小的条带卷。

提示： 4块物理磁盘中，只有5GB的空间是条带化的，剩余的空间还是常规的磁盘空间。为什么呢？条带化RAID 0不是需要至少两块物理硬盘么？这就是卷管理软件的优越性了。我们上文提过，卷管理软件将物理磁盘划分为PP和LV，所以有了更加细粒度的存储单位，条带化可以在这些LV之间进行，而其他LV不受影响。

创建RAID 5格式的卷。

- `vxassist -g DataDG make DataVolC 5g layout=RAID 5`

注意： RAID 5至少需要3块盘，否则不能成功。因为两块盘的RAID 5，还不如做 RAID 1。但是3块盘的RAID 5不能获得并发IO性能。

创建镜像卷（RAID 1）。

- `vxassist -g DataDG make DataVolD 5g layout=mirror`

创建RAID 10卷。

- `vxdg init RAID 10dg disk1 disk2 disk3 disk4`

创建磁盘组

- `vxassist -g RAID 10dg RAID 10vol 5g layout=mirror-stripe`

创建RAID 01卷。

- `vxdg init RAID 01dg disk1 disk2 disk3 disk4`

创建磁盘组。

- `vxassist -g RAID 01dg RAID 01vol 5g layout= stripe-mirror`

3. 创建文件系统并使用

- `mkfs -F vxfs /dev/vx/rdisk/DataDG/DataVolA`
- `mount -F vxfs /dev/vx/dsk/DataDG/DataVolA /mnt`

以上例子将卷DataVolA格式化成VxFS（Veritas公司的文件系统）格式，然后挂载于/mnt目录下，执行命令`cd /mnt`之后，就可以读写这个卷的内容了。

4. 动态扩大和缩小卷

将卷空间增加到10GB。

- `vxassist -g DataDG growto DataVolA 10G`

更改之后，卷的容量将会变成10GB。或者用`vxresize`命令。

- `vxresize -g DataDG DatavolA 10G`

将卷容量增加10GB。

- `vxassist -g DataDG growby DataVolA 10G`

或者用`vxresize`命令：

- `vxresize -g DataDG DatavolA +10G`

这样，更新之后卷的容量将在原来的基础上增加10GB大小。

5. 文件系统动态扩容

卷扩容之后，只是在卷的末尾增加了一块多余空间。这块空间如果没有文件系统的管理就无法存放文件，所以必须让文件系统将这块多余的空间利用起来。

- `fsadm -F vxfs -b 10240000 -r dev/vx/rdisk/DataDG/DataVolA /mnt`

6. 文件系统缩小

如果决定将某个卷缩小以省出更多空间，则在缩小卷空间之前，必须缩小文件系统的空间。也就是说，被裁掉的卷空间上存放的数据，需要转移到卷剩余的空间上存放，所以剩余空间必须足够，以便容纳被裁掉空间中的数据。

- `fsadm -F vxfs -b 5120000 -r dev/vx/rdisk/DataDG/DataVolA /mnt`

以上命令将这个卷上的文件系统缩小至5GB大小。剩余的5GB没有数据，可以被裁剪掉。

7. 卷容量缩小

在缩小了文件系统之后，卷容量方可缩小。

- `vxassist -g DataDG shrinkto DataVolA 5G`
- `vxresize -g DataDG DataVolA 5G`

上面的两个命令均可以使DataVolA卷的容量变为5GB。

- `vxassist -g DataDG shrinkby DataVolA 5G`
- `vxresize -g DataDG DataVolA -5G`

上面的两个命令均可以使DataVolA卷的容量在原来的基础上缩减5GB。

8. 从磁盘组中移除磁盘

若想从磁盘组中移除一块或者几块物理磁盘，则必须先将待移除物理磁盘上的数据转移到磁盘组中的其他物理磁盘的剩余空间中，这个动作通过下面的命令完成。

- `vxevac -g DataDG DataDG04 DataDG03`

上面的命令将disk4中的数据转移到disk3上。除了容量改变之外，不会影响卷的其他信息。

- `vxdg -g DataDG rmdisk disk4`

上面的命令将已经没有数据的disk4物理磁盘从磁盘组DataDG中移除（逻辑移除）。

- `vxdiskunsetup -C Disk4`

上面的命令将disk4物理磁盘从整个VxVM管理模块中注销。

5.6 大话文件系统

5.6.1 成何体统——没有规矩的仓库

话说这一天，老道闲来无事，在后山溜达。他走到了武当的粮库门口，发现这里堵了一大帮人。老道上前一问，原来这些人都是各个院来领取粮食的。只见他们一拥而上，进入仓库就各自找自己的房间去搬粮食。老道一看，怎么这么乱呢？就不能有个顺序么？

他向其中一个小道打听了一下，这才知道，造成这种乱七八糟进入粮库搬粮食局面的原因，是因为当初没有好好规划仓库。上个月，各个院从山下各自运了粮食上来，当时的政策是大家各自进入仓库，自己找房间放自己的粮食，自己找了哪些房间放粮食，自己记住了。到取粮食的时候，大家根据自己记录的房间来进入取粮。这个政策看似没什么可非议的，实则不然。如今山下粮食供应紧张，造成大家各顾各的，没有顺序，岂能不乱？老道进入粮仓一看，眼前一片狼藉！土豆、西红柿洒落得满地都是。这间房放这样，那间房放那样，就不能顺序地堆放粮食蔬菜？成何体统！！

提示： 在早期的计算机系统中，每个程序都必须自己管理磁盘，在磁盘中放自己的数据，程序需要直接和磁盘控制器打交道。有多少个程序要利用磁盘，就有多少个和磁盘交互的驱动接口。

老道摇了摇头，得想个办法彻底解决这个问题。老道回到了书房，闭目思索。首先大家不能都堵在门口，那么必须让他们排起队来。其次，每个人各顾各，自己记录自己用了哪间房子，一个是浪费，另一个

是容易造成冲突。一旦某个人记错了，就会影响其他人。那么就应该只让一个人记录所有人的信息，他自己不会和自己冲突。同样这个人也要充当一个门卫的作用，接待来取粮或者送粮的人，让他们按一定的顺序来运作。

最终决定就应该是这样的：找一个人，这个人的职责就是接待来取粮或者送粮的人，把要取的或者要送的粮食的名称和数量等信息先登记在这个人的一个本子上，然后由这个人来合理地选择仓库中的房间，存放或提取登记在案的粮食，而且提取或放入粮食之后要将本子上的记录更新，以便下次备查。嗯，这么做就好多了，哈哈哈哈！这天晚上的北斗七星，光芒格外耀眼。

5.6.2 慧眼识人——交给下一代去设计

第二天，老道亲自挑选了一位才思敏捷、内向稳重、善于思考的道士来担任这个重要的角色。让他和库管员一起完成管理粮库的工作，给他起了一个职称，叫做理货员。并且将自己的想法告诉了这位道士，让他当晚就考虑出一套符合这个思想的方法，还可以做出改进意见。

就这样，又过了一晚。第三天，这位道士上任了。一大早，张老道就在暗中观察。这时候，一个送粮食的人来了，他带了1024斤土豆和512斤白菜。这人还是按老习惯，上来就往仓库闯。小道士截住了他：“道长且慢！请问您送的是什么蔬菜？”那人道：“土豆和白菜！”小道士又道：“土豆多少斤？”答曰：“土豆1024斤。”（上面这个过程就是应用程序和FS的API交互的过程）。小道士笑道：“道长尽可放心将土豆交于我，我自当为您找房间存放。”然后小道士到仓库中找了两个空房间，每个房间放了512斤土豆。并在本子上记录：“土豆1024斤房间1—2。”接着他就命令库管员来搬运货物到相应的房间。

道士给每个库区都预备了一个记录本。小道士不关心具体房间到底在仓库哪里，怎么走才能达到，这些事情统统由库管员来协调。小道士同样也不关心来送货物的人到底送的是什么货物。如果送粮的人告诉他，请给我存放rubbish 1000斤，道士眼都不眨照样给他存放。一旦仓库的房间都满了，小道长再次命令库管员搬运货物时，库管员就会告诉他，已经没有房间了。那么道长就告诉来存放货物的人：“对不起，空间不足。”

用同样的方法，小道士将那人的白菜，也放到了一间房中，记录下：“白菜 512斤 房间3”。然后向那人说到：“这位道长，您下次来取的时候，直接向我说要某厨房存放的土豆多少斤就可以了，我会帮您找到并取出。”那人非常满意地离去了。接着又有很多人 also 来送取冬瓜、南瓜、西瓜、大米、面粉等粮食蔬菜，小道士一一对应，有条有理。小道士也专门给自己在每个库区中预留了几间房，用于存放他那一本本厚厚的记录。老道一旁看了，频频点头，“嗯，前途无量，前途无量啊，啊哈哈哈哈哈！！！”

过了几天，张真人又来探查。此时只见有个人一下送来10000斤大米。小道长开始只是表示吃惊，并没有多想，仍旧按照老办法，记录：“大米10000斤，房间4—4096”。接着又来了一位要存放65535斤小麦的。这下可苦了小道士了，把他累得够呛。随着全国粮食大丰收，存粮数量动辄上万斤。这让小道士苦不堪言，他决定思考一种解决方法。第二天，小道长将每8个房间划分为一个逻辑房间，称作“簇”。第一簇对应房间1、2、3、4、5、6、7、8，第二簇对应房间9、10、11、12、13、14、15、16，依此类推。这样道士记录的数字量就是原来的八分之一了。比如4000斤粮食，只需记录“簇1”就可以了。老道心中暗想，“嗯，不错，我没看错人！”这一年，因为大丰收，粮食降价了。农

民丰产不丰收，很多农民打算第二年不种粮了，改做其他小生意。

5.6.3 无孔不入——不浪费一点空间

第二年，果然不出张老道所料，全国粮食大减产，价格飞涨，全面进入恐慌阶段。张真人悬壶济世，开仓放粮，平息物价。这一举动受到了老百姓的称赞和感激，但也招致了一小部分奸商的忌恨。

放粮消息宣布之后，山下老百姓都排队来武当买粮。这可忙坏了理货员道士，连续几天没休息，给老百姓取粮食。一个月之后，武当粮库存粮已经所剩无几，张老道和众院道士每天省吃俭用，为的是给老百姓多留点存粮。

大恐慌的一年，终于熬过去了。农民一看粮食价格那么高，第三年又都准备种粮了。不出意料，这一年粮食又得丰收！张老道提前考虑他的粮库在这一年的使用问题了，他叫来理货员道士，让他回去考虑一个问题：经过了去年的折腾，仓库中的存货是零零散散，乱七八糟，为了准备这一年大量粮食涌入仓库，必须解决这个问题，让他回去考虑解决办法。其实张老道早就在心里盘算出了解决办法了。

第二天，理货员趁人少的时候，就命令库管员：“请帮我把房间XXXX的货物移动到房间XXXX处，请帮我把房间XXX的货物移动到房间XXX处……。”

这可累坏了库管员。但是经过几个时辰的整理之后，仓库里的货物重新变得连续，井井有条。老道称赞说：“不错！继续努力！”

这天晚上，小道长也没闲着，他继续思考，今天是有时间整理货物，如果一旦遇到忙的时候，没有时间整理货物，那麻烦就大了，得想

一种一劳永逸的办法。有些人来送完粮食之后，第二天就来取了，这个真是头疼了。因为我都按照顺序将每个人的粮食连续存放到各个簇中，他一下取走了，对应的簇就空了。如果再有人来，他带的货物数量如果这个空簇能存下还好，可以接着用。如果存不下呢？还得找新的连续空簇来存放。如果这种情况出现太多，那么整个仓库就是千疮百孔，大的放不下，小的放下了又浪费空间……真头疼。他冥思苦想，最后终于想出一个办法。

一早仓库还没有开门的时候，小道长就来了，他把所有记录本都拿了出来，进行修改。他原本对每个来送货的人，都只用一条简单记录来描述它，描述中包含3个字段：名称、大小和存放位置。比如冬瓜 10000斤 簇1-3。此时仓库中，虽然总空余空间远远大于10000斤的量，但是已经没有能连续地放入10000斤大小的簇空间，那么这个货物就不能被放入仓库，而这是不能容忍的一种浪费。有一个办法，就是上面说过的，找空闲时间来整理仓库，整理出连续的空间来。这次小道长想出了另一个方法，就是将货物分开存放，并不一定非要连续存放在仓库。因为仓库已经被逻辑分割成一簇（8个房间）为最小单位存放货物。那么就可以存在类似这样的描述方式：冬瓜 10000斤 簇2、6、19。也就是说这10000斤的冬瓜是分别被按顺序存放在仓库的2号簇、6号簇和19号簇中的。取出的时候，需要先去2号取出货物，再跨过3个簇去6号，再跨过13个簇去19号。都取出后再交给提货人。这样确实慢了点，但是完美地解决了空间浪费的问题。

5.6.4 一箭双雕——一张图解决两个难题

粮食大丰收果然又被张老道猜中了，这次小道长是应对自如，一丝不乱。老道啧啧称赞！但是老道却从小道士的记录中，又看出了一些问

题，他告诉小道士，要继续思考更好的解决办法。小道士心很灵，他知道这个方法确实解决了问题，但是有缺陷，会有后患，只不过现在的环境并没有显示出来。这天晚上，小道在仓库睡觉，没有回去。

提示： 看着他那些记录，只见上面一条一条、一行一行的，却也比较有条理。但是仔细一看发现，每一条记录的最后一个字段，也就是描述货物存放在哪些簇的那个字段，非常凌乱，因为每个人送来的货物数量不一样，那么就注定这个字段长短不一，显得非常乱。现在记录不是很多，但记录一旦增多，每次查询的时候就很不方便。而且要找一个未被占用的簇，需要把所有已经被占用的全找出来，然后才去选择一个未被占用的簇，分配给新的货物存放。这个过程是非常耗时间的，货物少了还可以，货物一多，那可就费劲了。“嗯，张真人让我继续思考，确实是有道理的，这两个隐患，确实是致命的，尤其是第二个。得继续找新方法。”

提示： 小道士继续思考。第一个问题，要想解决长短不一的毛病，最简单的就是给他一个定长的描述字，这仿佛是不可能的，有的需要1个簇就够了，有的却需要10个甚至100个，如果把这需要10个簇的和需要1个簇的，都用1个簇来描述，那么确实非常漂亮了，记录会非常工整。

想到这里，小道士累了，想出去走走。他溜达到一个路口，看见路口上有路标牌，上面写着：“去会客厅请走左边，去习武观请走右边。”小道士顺着路标指向，走了右边，然后又遇到一个路标：“去习武观请走左边，下山请走右边。”道士走了左边，最终来到了习武观。他看着习武观正中央的那个醒目的“道”字，忽然眼前一亮！

他迅速原路返回到粮库，拿出记录本，将其中一条记录改为：“冬

瓜 10000斤 首簇1”。每条记录都改成这种形式，也就是只描述这个货物占用的第一个簇的号码，这样完美解决了记录长短不一的问题，那么后续的簇呢？只知道首簇，剩余的不知道，一样不能全部把货物取出。

所以小道士参照路标的形式，既然知道了首簇号，那么如果找到首簇，再在首簇处作一个标记，写明下一个簇是多少号，然后找到下一个簇取货，然后再参照这个簇处的路标，到下下个簇处接着取货，依此类推，如果本簇就是这批货物的最后一簇，那么就标识：“结束，无下一簇”。比如：“冬瓜 10000斤 首簇1”这个例子，先把4096斤冬瓜放到簇1中，然后在簇1的门上贴上一个标签：“簇10”，这就表明下一簇是10号簇。继续向10号簇中存入4096斤冬瓜，此时还剩808斤冬瓜没放入，还需要一个路标，于是在10号簇的门上再贴一个标签：“90号”。然后去90号簇放入剩下的808斤冬瓜。

第二天，张老道继续来视察。老道一看他的记录，不由地一惊！“一个晚上就想到了这种绝妙方法。嗯，此人大有前途！”老道频频点头称赞。然后老道进仓库查看，一看有些簇的门上，贴着标签，老道立即明白了小道长的做法，对小道说：“孩子，不错，但是还需要再改进！”

小道心里盘算，“嗯，这个方法是解决了第一个问题，但是每个簇门上都贴一个标签，这样是不太像样。而且寻找未被占用的簇的效率还是那么低，还是需要把所有已经占用的簇找出来，再比对选出没有使用的空簇。而且我这么一弄，找空簇的效率比原来还差了，因为原来已经使用的簇都会被记录在货物描述中的字段中，现在把这个字段缩减成一个字了，这样每次找寻的时候，还得去仓库中实际一个门一个门地去抄下已经使用的簇，还不如直接在本子上找来得快。这个问题得解决！”

思考：既然要拿掉贴在门上的标签，那么就必须找另外一个地方存放标签，所以只能存放到我的记录本上。可是各个簇的路标我都记录在本子上，用一个什么数据结构好呢？货物描述那三个字段肯定不能再修改了，那样已经很完美了，不能破坏它。那么就需要再自己定义一个结构来存放这些路标之间的关系，而且每个货物的路标还不能混淆，混了就惨了。他在纸上写写画画，不知不觉把整个仓库的簇画出来了，从第一个簇，到最后一个簇，都用一个方格标识，然后他参照“冬瓜，10000斤，首簇1”这个例子，下一簇是簇10，那么他在簇1的格子上写上了“簇10”，然后他找到第10个格子，也就是代表簇10的格子，在簇10格子里面写上“簇90”，也就是10号簇的下一簇路标。然后继续找到90号簇，此时他在这个格子里写上“结束”。接着他又举了几个例子，分别画了上去。就这么逐渐睡着了。

第二天早晨，小道士迷迷糊糊地起来了，只见张道长已经在他的面前，带着赞许的笑容。“孩子，你累了，不错不错，你终于把所有问题都解决了啊！”张老道摸着小道士的头，称赞地说道。小道士还不知道是怎么回事呢，他告诉张老道说，他还没想出来呢。老道大笑说：“哈哈哈哈哈，你看看你画在纸上的图，这不是已经解决了么？哈哈哈哈哈。”说完老道扬长而去。

小道士一头雾水，看着那张画，这才想起了昨晚的思考。“对啊，这张图不就行了么？这就是我所要找的数据结构啊！”接着，小道士把图重新画了一张，工工整整地夹在了记录本里面。这时，来了一个取货的人，他告诉小道士说：“二库区，南瓜，10000斤。”道士说：“稍等。”然后立即查询二库区的记录本，找到南瓜的记录，发现首簇是128。然后立即到那张图上找到第128号簇所在的格子，发现上面写的

是“簇168”。继续找到第168号格子，上面写的是“簇2006”。立即找到第2006个格子，只见上面写的是“结束”。然后他通知库管员：“请将第128、168、2006三个簇的货物提取出来给我。”不一会儿，货物到了，交货签字。小道士恍然大悟，“太完美了！！”

紧接着，又来了一个存货的人，他有西瓜500斤要存放到1库区。小道士立即查看那张图，一目了然。只要格子上没有写字的就是空簇，就可以用来存放货物。所以道士立即找到一个空着的50号簇来存放这500斤西瓜。存放完毕之后，在对应的这个格子上写上“结束”，因为500斤的数量一个房间就够了，更不用说一个簇了（最多8个房间）。接着也在1库区的记录本上增加一条记录“西瓜 500斤 首簇50”。

道士发现，第二个问题也就是查找未被使用的簇的问题，自从有了这张图，就自然解决了。道士非常兴奋，同时也佩服张真人，是他引导着自己一步一步解决问题的。

5.6.5 宽容似海——设计也要像心胸一样宽

随着仓库业务的不断成熟，小道士的技能越来越熟练，他开始考虑描述货物的三个字段：名称、数量、存放的第一个簇。随着国民生产力水平不断提高，各种层出不穷的产品被生产出来，它们有些具有一些奇特的属性。所以小道士准备增加字段来表述一件货物更多的属性，比如送货时间、只读、隐藏等各种花哨属性。同时，那张图也不能满足要求了，因为随着生产力发展，仓库每平方米造价越来越低，武当决定扩大仓库容量。这样仓库中所包含的簇数量就大大增加了，甚至成几何数量级增长，所以簇号码越来越大，甚至超过了亿。要记录这么多位的数字，本来那个小格子就写不开了，所以需要增大格子的宽度，以便能写下更多的数字位数。以前每个格子是2字节（16位）长度，现在扩展到

了4字节（32位）。而据传江湖上另一位大侠已经将格子的宽度扩展到了128位。

这位小道长姓字名谁？因为当时张真人收留他的时候，发现他身板有点软，不适合练武。但思维敏捷，适合练心法，所以给他一个道号叫做微软。

就这样，仓库又运作了两年。

5.6.6 老将出马——权威发布

仓库存储容量不断增加，仓库管理技术方面却并没有什么进步，还是沿袭两年前那一套运作模式。这显然已经不适应现代仓库了，所以造成入库等待、处理速度逐渐变慢等一系列的问题。张真人决定跟上时代，要研究出一套新的仓库运作模式，并且定义出一个规范，让全天下的仓库都沿袭这个规范来运作。张老道先仔细考察了微软道士的运作模式，然后根据现代仓库管理的特点，提出了一系列的解决方案。

现代仓库管理要求入库出库速度快，由于在仓库硬件方面提高很快，有了更加新式的传送带和机器人等机器，所以大大提高了操作简化度，减轻了库管员的负担。库管员只需要阅读机器的随机手册（驱动程序）便可以轻松地完成操作。与此同时，对于理货员这块技术并没有什么新的突破，因为理货这块主要靠好的算法，并不需要硬件支持，除了那些记录本之外。而从仓库中取出记录本的速度，由于库管员操作迅速，所以也不在话下。关键就看理货算法了。这是任何硬件都不能解决的问题。

首先张老道通过观察、记录，发现一般货物就算是存放 to 不连续的簇中，这些簇往往也是局部连续的，比如1、2、3、5、6、7、100、

101、102，其中1、2、3就是局部连续，5、6、7也是，100、101、102也是。而不太可能出现一个货物占用了1、56、168、2008簇这种情况。如果此时不是一个簇一个簇地去找路标，而是一段一段地去找，这样会节约很多时间和精力。比如簇段1~3，簇段5~7，簇段100~102。这样就大大简化了路标。还有其他的一些改进方式，如直接将一些小货物存放到它们的描述记录中（驻留文件）。只有描述记录中放不下时，才到仓库其他区域找一些簇来存放，然后记录这些簇段。

微软道士将他的记录本上的信息，称为Metadata，即元数据，也就是用来描述其他数据是怎么组织存放的一种数据。如果记录本丢失，那么纵然仓库中货物完好无损，也无法取出。因为已经不知道货物的组织结构了。

张真人最后把微软道士实现的一共三种仓库运作管理模式，分别叫做FAT16、FAT32和NTFS，并取名为小道藏龙。

5.6.7 一统江湖——所有操作系统都在用

后来张老道把这套管理模式移植到了磁盘管理上，这就是轰动武林的所谓“文件系统”。对应仓库来说，送货人送来的每一件货物都称作“文件”。取货时，只要告诉理货员文件名称、所要取出的长度及其他一些选项，那么理货员就可以从仓库中取出这些数据。

在一个没有文件系统的计算机上，如果一个程序要向磁盘上存储一些自己的数据，那么这个程序只能自己调用磁盘控制器驱动（无VM的情况下），或者调用VM提供的接口，对磁盘写数据。而写完数据后，很有可能被其他程序的数据覆盖掉。引入文件系统之后，各个程序之间都通过文件系统接口访问磁盘，所有被写入的数据都称为一个文件，有

着自己的名字，是一个实体。而且其他程序写入的数据，不会将其他人的文件数据覆盖掉，因为文件系统会计算并保障这一点。

除此之外，不仅张真人的NTFS文件系统取得了巨大的成功，适应了现代的要求。与此同时，少林的雷牛方丈也创造出了其他的文件系统，比如EXT一代、二代、三代和JFS等文件系统。一时间文件系统思想的光环是照耀江湖！！

5.7 文件系统中的IO方式

那么，有了文件系统之后，整个系统是个什么架构？

图5-75为Windows系统的IO简化流程图。

图5-75 Windows系统的IO流程图

图中的IO Manager是操作系统内核的一个模块，专门用来管理IO，并协调文件系统、卷、磁盘驱动程序各个模块之间的运作。整个流程解释如下。

- 某时刻，某应用程序调用文件系统接口，准备写入某文件从某个字节开始的若干字节。
- IO Manager最终将这个请求发送给文件系统模块。
- 文件系统将某个文件对应的逻辑偏移映射成卷的LBA地址偏移。
- 文件系统向IO Manager请求调用卷管理软件模块的接口。

卷管理软件将卷对应的LBA地址偏移翻译映射成实际物理磁盘对应的LBA地址偏移，并请求调用磁盘控制器驱动程序。

- IO Manager向磁盘控制器驱动程序请求将对应LBA地址段的数据从内存写入某块物理磁盘。

文件系统的IO包括同步IO、异步IO、阻塞/非阻塞IO和Direct IO。

(1) 同步IO： 同步IO是指程序的某一个进程或者线程，如果某时刻调用了同步IO接口，则IO请求发出后，这个进程或者线程必须等待IO路径上的下位程序返回的信号（不管是成功收到数据的信号还是失败的

信号)。如果不能立刻收到下位的信号;则一直处于等待状态,不继续执行后续的代码,被操作系统挂起,操作系统继续执行其他的进程或者线程。

而如果在这期间,倘若IO的下位程序尚未得到上位程序请求的数据,此时IO路径上的下位程序又可以选择两种动作方式:第一是如果暂时没有得到上位程序请求的数据,则返回通知通告上位程序数据未收到,而上位程序此时便可以继续执行;第二种动作则是下位程序也等待它自己的下位程序来返回数据,直到数据成功返回,才将数据送给上位程序。前者就是非阻塞IO,后者就是阻塞IO方式。

同步+阻塞IO是彻底的堵死状态,这种情况下,除非这个程序是多线程程序,否则程序就此挂死,失去响应。同理,异步+非阻塞的IO方式则是最松耦合的IO方式。

(2) 异步IO: 异步IO请求发出后,操作系统会继续执行本线程或者进程中后续的代码,直到时间片到时或者因其他原因被挂起。异步IO模式下,应用程序的响应速度不会受IO瓶颈的影响,即使这个IO很长时间没有完成。虽然应用程序得不到它要的数据,但不会影响其他功能的执行。

基于这个结果,很多数据库在异步IO的情况下,都会将负责把缓存Flush到磁盘的进程(Oracle中这个进程为DBWR进程)数量设置成比较低的数值,甚至为1。因为在异步IO的情况下,Flush进程不必挂起以等待IO完成,所以即使使用很多的Flush进程,也与使用1个进程效果差不多。

异步IO和非阻塞IO的另一个好处是文件系统不必立刻返回数据,所

以可以对上层请求的IO进行优化排队处理，或者批量向下层请求IO，这样就大大提升了系统性能。

(3) Direct IO: 文件系统都有自己的缓存机制，增加缓存就是为了使性能得到优化。而有些应用程序，比如数据库程序，它们有自己的缓存，IO在发出之前已经经过自己的缓存算法优化过了，如果请求IO到达文件系统之后，又被缓存起来进行额外的优化，就是多此一举了，既浪费了时间，又降低了性能。对于文件系统返回的数据，同样也有这个多余的动作。所以文件系统提供了另外一种接口，就是Direct IO接口。调用这种接口的程序，其IO请求、数据请求以及回送的数据将都不被文件系统缓存，而是直接进入应用程序的缓存，这样就提升了性能。此外，在系统路径上任何一处引入缓存，如果是Write Back模式，都将带来数据一致性的问题。Direct IO绕过了文件系统的缓存，所以降低了数据不一致的风险。

提示：关于更详尽的系统IO论述，请参考本书后面的章节。

第6章 阵列之行——大话磁盘阵列

■ 磁盘阵列

■ SCSI

■ LUN

■ 前端/后端

两三块磁盘做RAID 0或1，四五块磁盘做个RAID 3、4、5是小事一桩，不过太没魄力。要玩就弄个几十块盘，那才过瘾。这不，有人发明了专门装这些磁盘的大柜子，我们这就去看看这柜子是怎么回事儿吧。

退隐江湖——太累了，该歇歇了

自从张真人创立了降龙三掌之后，江湖各门各派争相修炼，商人不断推出基于降龙掌的新商品。江湖上浮躁之气再次袭来，很少有人去钻研底层功夫了，都是拿来就用，不思进取。几十年过去了，张老道已经成了头发苍白的老人。

这天晚上，人少星稀。唯独天上的北斗七星，光芒还是那么灿烂，仿佛已逝去百年的七星大侠，还在天上苦苦钻研。

张真人如今也已经是白发苍苍，可是知己已不在。一百年来，江湖上为了利益你争我抢，反目成仇，打打杀杀。呜呼哉！！！难道这个江

湖真要从此衰败么？张老道失望至极。

闻道

尘世浮华迷人眼，
梦中情境亦非真。
朝若闻道夕死可，
世间何处有高人？

第二天，张真人对外宣布，他从此退隐江湖，不再参与江湖事。瞬间，整个江湖就像地震了一样，人们没有了主心骨，都不知道该干什么好了。打打杀杀的也不打了，商人也没得吹了。很多商人纷纷上武当来游说张真人，让他出山，包荣华富贵，都被张真人一一回绝了。江湖又恢复了以往的平静，只是这平静似乎预示着一场更加猛烈的暴风雨即将来临。

前仆后继——后来者居上

话说有位少年，自幼好钻研和寻根问底，被人称作“隔一路”。此人不善于口头表达，不会忽悠，但是如果世界只剩下他一个人，那么他便会爆发出神奇的力量。由于内向的性格，他吃了不少亏，但他依然我行我素，并不在乎别人的议论和猜忌甚至是诋毁。这位少年名为无忌。他实际上也确实是无所畏惧，明知山有虎，偏向虎山行，用天真和执著去挑战世俗，跌倒了大不了重来。

既然选择了这条路，就要把它走完。孤独和压迫给了他巨大的动力，每天晚上都在刻苦学习。他学习IO大法和磁盘大挪移，学习七星北斗阵和降龙大法。虽然他并没有实践过这些知识，但是依然有一股力量

促使他不断地学习钻研。

6.1 初露端倪——外置磁盘柜应用探索

无忌已经充分掌握了前人留下的心法口诀。在不知道该做什么的时候，他突然有了一个想法。虽然按照七星大侠的RAID方式，可以将多块磁盘做成逻辑盘，但是普通的服务器或者PC机箱里面，也就安装两 three 块磁盘，空间就满了。如果做很多块盘的RAID，把磁盘都放到机箱里面肯定不行，得想个办法来让机器可以带多块磁盘。

“拿出来，拿出来，全部都掏出来！”他找来一台机器，装了一块 Ultra 320 SCSI卡，这个卡只有一个通道，可以连接15块磁盘。但是15块盘怎么放入一个机箱呢？太困难了，所以必须把这些盘放到机箱外面。但是连线 and 电源问题又不好办。他索性找来一个箱子，把所有磁盘都放在这个箱子里。箱子有独立电源和散热系统，保障磁盘的稳定运行。接口方面，内部其实就是一条SCSI线缆，只不过将它做到了电路板上，然后在外面放一个接口，这个接口是用来连接主机上的SCSI卡的。如果主机上装的是不带RAID功能的SCSI卡，那么加电之后，主机会识别到磁盘箱中的所有磁盘。箱子中有多少磁盘，在OS磁盘管理器中就会显示多少块磁盘。如果主机上安装的是带RAID功能的SCSI卡，那么可以用这个RAID卡先来对认到的多块磁盘做一下RAID，划分出逻辑盘，这时OS识别到的就是逻辑磁盘，而不会认到箱子中的物理磁盘。

这种简单的磁盘箱如图6-1所示，无忌给它取了个学名，叫做“JBOD”，也就是Just a Bound Of Disks，“只是一串磁盘”，这个描述非常形象。无忌立即将这个做法公布了出去，没想到大受欢迎，一时间各个厂家争相生产这种磁盘柜，在市场上卖得很火。

图6-1 JBOD磁盘阵列示意图

6.2 精益求精——结合**RAID**卡实现外置磁盘阵列

思考：能否把RAID功能做到磁盘箱中，因为如果要调整RAID的话，还需要重启主机等，会影响主机应用。如果做到了磁盘箱中，那么在主机上就不需要做什么，只要在磁盘箱中做完之后连接到主机，主机重启之后或者不用重启就能认到新逻辑盘了。

经过多次实验，终于做成了一个设备。少年把这种自带RAID控制器的磁盘箱叫做“磁盘阵列”。自此在江湖上有了一个不成文的规定，凡是JBOD都叫做磁盘柜，凡是自带RAID控制器的盘柜就叫做磁盘阵列或者盘阵。盘柜和盘阵，前者只是一串外置的磁盘，而后者自带RAID控制器。图6-2为JBOD磁盘柜实物图。

图6-2 JBOD磁盘柜实物图

盘阵是在盘柜的基础上，将内部的磁盘经过其自带的RAID控制器的分分合合，虚拟化成逻辑磁盘，然后经过外部SCSI接口连接到主机上端的SCSI接口。此时，整个盘阵对于主机来说，就是主机SCSI总线上一个或者多个设备，具有一个或者多个SCSI ID。所有逻辑磁盘都以LUN的形式呈现给主机。

如图6-3所示，盘阵中的SCSI控制器在逻辑上有两个部分，右边的S2控制器连接了一条SCSI总线，上面有若干磁盘。左边的S1控制器同样也连接了一条SCSI总线，但是上面只有两个设备：一个就是主机SCSI控制器，另一个就是它自己。

图6-3 带RAID控制器的磁盘阵列示意图

毫无疑问，在左边的SCSI总线上，盘阵SCSI控制器是作为Target模式，被主机SCSI控制器操控，处于被动地位；在右边的SCSI总线上，盘阵的S2 控制器成了Initiator模式，它在右边总线上占据主动权，拥有最高优先级，而各个磁盘均为SCSI Target，受控于Initiator。当然S1和S2不一定就是两块物理上分开的芯片，很有可能就是一块单独的芯片逻辑地分成两个部分。甚至有可能将RAID芯片和SCSI控制器芯片全部集成到一个大芯片中。

图6-4所示的是一个SATA盘阵控制器的主板示意图。

图6-4 一个可以连接16块SATA磁盘的小型RAID控制器主板

图6-5所示的是一个小型盘阵控制器的内部实物图。

图6-5 一个小型控制器实物图

图6-6所示的是一台盘阵的磁盘插槽实物图。

图6-6 磁盘插槽、背板

图6-7所示的为这台盘阵的电源模块插槽。

图6-7 电源模块插槽

6.3 独立宣言——独立的外部磁盘阵列

主机由于肚量太小容不下想法太多的磁盘！终于磁盘从主机内部跑出来了，磁盘们在外置的大箱子里，在RAID控制器的带领下，欣欣向荣，勇往直前！

磁盘和控制器发布了独立宣言，彻底摆脱了主机的束缚，成为与主机对峙的一个独立的外部设备。从此以后，存储技术才真正的成为一个独立的庞大学科，并不断发展壮大。本书后面的章节会介绍更多的存储技术，包括存储网络和网络存储。

1. 前端和后端

对于盘阵来说，图6-3中RAID控制器的左边就称为“前端”，右边则称为“后端”。面向主机对外提供服务的就叫前端，面向自己管理的磁盘用于内部管理而外部不需要了解的部分就叫做后端。同样，对于主机来说，它的SCSI适配器反而成了后端，而以太网卡可能变成了前端。因为对于主机来说，直接面对外部客户机的是以太网，而管理磁盘的工作不必对客户说明，所以变成了后端。

2. 内部接口和外部接口

对于盘阵来说，还有一个内部接口和外部接口的概念。内部接口是指盘阵RAID控制器连接其内部磁盘时用的接口，比如可以连接IDE磁盘、SCSI磁盘、SATA磁盘和FC磁盘等。外部接口是指盘阵控制器对于主机端，也就是前端，提供的是什么接口，比如SCSI接口、FC接口等。内部接口可以和外部接口相同，比如内部用SCSI磁盘，外部也用

SCSI接口连接主机，这种情况也就是图6-3中所示的情况。

内外接口也可以不同，比如内部连接IDE磁盘，外部却用SCSI接口连接主机（仅限于盘阵，盘柜必须内外接口一致）。盘阵控制器是一个虚拟化引擎，它的前端和后端可以不一致，它可以向主机报告其有多少LUN，尽管内部的磁盘是IDE的。

3. 多外部接口

同时，我们也不要被盘阵上为什么可以有多个外部SCSI接口而感到困惑。有多个接口是为了连接多台主机用的。每个由盘阵RAID控制器生成的逻辑磁盘，可以通过设置只分配（Assign/Map）到其中一个口，比如LUN1被分配到了1号口，那么连接到2号口的主机就不会看到这个LUN。也可以把一个LUN同时分配（或叫做Map，映射）到两个口，那么两台主机能同时识别出这个LUN。让两台主机同时对一个LUN写数据，底层是允许的，但是很容易造成数据的不一致，除非使用集群文件系统，或者高可用性系统软件的参与。

4. 关于LUN

LUN是SCSI协议中的名词，我们前面也描述过。LUN是SCSI ID的更细一级的地址号，每个SCSI ID（Target ID）下面还可以有更多的LUN ID（视ID字段的长度而定）。对于大型磁盘阵列来说，可以生成几百甚至几千个虚拟磁盘，为每个虚拟磁盘分配一个SCSI ID是远远不够用的。因为每个SCSI总线最多允许16个设备接入（目前32位SCSI标准最大允许32个设备）。要在一条总线上放置多于16个物理设备也是不可能的，LUN就是这样一个次级寻址ID。磁盘阵列可以在一个SCSI ID下虚拟多个LUN地址，每个LUN地址对应一个虚拟磁盘，这样就可以在

一条总线上生成众多虚拟磁盘，以满足需求。

后来，人们把硬件层次生成的虚拟磁盘，统一称为“LUN”，不管是不是在SCSI环境下，虽然LUN最初只是SCSI体系里面的一个概念。而由软件生成的虚拟磁盘，统一称为“卷”，比如各种卷管理软件、软RAID软件等所生成的虚拟磁盘。

有些盘阵配有液晶操控面板，如图6-8所示。而有些低端的盘阵更是在液晶面板周围加上了按钮，用来对盘阵进行简单快速的配置，比如查看磁盘状态、设置RAID、划分逻辑磁盘等。这种方式极其简化了配置操作，一般用户通过阅读说明书就可以完成配置。不过液晶屏幕比较小，能完成的功能不多，操作相比用鼠标要麻烦。所以一些盘阵提供了COM口或者以太网接口，可以用PC连接这些接口与盘阵通信，通过仿真终端或Web界面就可以对盘阵进行配置。

图6-8 一个带液晶面板的盘阵前视图

提示： 用户用PC与盘阵的COM口或专用于配置的以太网接口连接，完全是为了配置磁盘阵列的各种参数，而不是通过这些配置专用接口从磁盘阵列的磁盘上读写数据。

6.4 双龙戏珠——双控制器的高安全性磁盘阵列

如果盘阵内部只有一个控制器模块，那么会是一个SPOF（Single Point Of Failure），即单点故障点。所以一些高端的盘阵内部都有两个控制器，互为冗余。分配给其中一个控制器的LUN逻辑卷，可以在这个控制器因故障失效的时候，自动被另一个工作正常的控制器接管，继续处理针对这个LUN的读写请求。两个控制器平时都管理各自的LUN，一旦发现对方故障，那么就会自动将所有LUN都接管过来。

因为如此，两个控制器之间需要相互通信，通告对方自己的状态以及交互一些其他的信息。两个控制器之间可以用PCI总线连接，也可以用厂商自己设计的总线来连接，没有统一标准。至于交互信息的逻辑和内容，更是因品牌而不同，而没有标准来统一它们。

为了避免单点故障，给盘阵安装一个额外的控制器，这个控制器和原来的控制器在它们后端共享一条或者多条磁盘总线。两个控制器可以使用Active-Standby方式，也可以使用Dual-Active的互备方式连接。

1. Active-Standby

这种方式又称HA（High Availability方式，高可用性），即两个控制器中同一时刻只有一个在工作，另外一个处于等待、同步和监控状态。一旦主控制器发生故障，则备份控制器立即接管其工作。

对于内部为SCSI总线的双控制器盘阵，在机头内部的一条SCSI总线中，两个控制器可以分别占用一个ID，这样剩余14个ID给磁盘使用。平时只有主控制器这个ID作为Initiator向除了备份控制器ID之外总线上

的其他ID（也就是所有磁盘的ID）来发送指令从而读写数据。

同时备份控制器与主控制器之间保持通信和缓存同步，一旦主控制器与备份控制器失去联系，那么备份控制器立即接管主控制器。同时为了预防脑分裂（见下文），备份控制器在接管之前需要通过某种机制将主控制器断电或者重启，释放其总线使用权，然后自己接管后端总线和前端总线。

提示： 主机端必须用两个SCSI适配器分别连接到盘阵的两个控制器上，才能达到冗余的目的，但是这样做主机端必须通过某种方式感知到这种HA策略并在故障发生时切换。目前，由于SCSI盘阵本身比较低端，可接入容量不大，所以没有双控制器的设计，以上文字只是对HA机制的一种描述。但是对于本书后面要讲述的FC盘阵来说，使用双控制器以及在主机端使用双FC适配卡是非常普遍的。

2. Dual-Active

顾名思义，这种双控制器的实现方式是两个控制器同时在工作，每个控制器都对所有后端的总线有通路，但是每个总线平时只被其中一个控制器管理，另一个控制器不去触动。可以将后端一半数量的总线交由一个控制器管理，另一半交由另外一个控制器管理。一旦其中一个控制损坏，则另外一个控制器接管所有总线。这种方式比Active-Standby方式高效很多。

3. 脑分裂（Split Brain）

这个词明显有点恐怖。设想一下，如果某时刻连接两个控制器之间

的通路出现了问题，而不是其中某个控制器死机，此时两个控制器其实都是工作正常的，但是两者都检测不到对方的存在，所以两者都尝试接管所有总线，这时候就是所谓的“脑分裂”，即同时有两个活动控制器来操控所有后端设备。这种情况是可怕的，类似精神分裂症。

如何预防这种情况呢？通常做法是利用一个仲裁者来选择到底使用哪一个控制器接管所有总线，比如用一两个控制器都能访问到的磁盘，控制器向其上写入自己的仲裁信息。一旦发生脑分裂，二者就参考这个磁盘，谁最后写入了信息就把控制权给谁。或者用一种电源控制器，一旦其中某个控制器要接管，那么不管对方是确实发生故障了还是正常的，这个控制器都会向电源控制器发送信号，让对方重启并进入Standby状态，这样就成功地预防了脑分裂。

接管了总线的控制器一般都会对总线上所有磁盘进行SCSI Reserve操作，即预订操作。总线上所有目标设备一旦被预订，它们便不再接受其他控制器的IO请求。SCSI 2标准中的SCSI Reserve不允许其他控制器读写被原有控制器预订的设备，但是SCSI 3中的Reserve策略有了一些灵活性，可以允许其他控制器对已经被预订的目标设备进行读IO，而写IO则被拒绝。

图6-9所示的是一双控制器盘阵机头示意图。

图6-9 双控制器磁盘阵列示意图

提示： 实际中，由于SCSI盘阵比较低端，一般没有这种设计模式的产品。

6.5 龙头凤尾——连接多个扩展柜

一条SCSI总线最多可以连接15块磁盘，为了这15块磁盘，大动干戈地赋予两个昂贵的RAID控制器，有点不值。为了把这两个控制器充分利用起来，榨取最后一滴性能，15块磁盘不够，那就再加。前面说过，一个控制器上可以有多个通道，一个通道下面就是一条SCSI总线，那么将盘阵的每个控制器上再多接一个或者两个通道，来充分发挥它的能力，这样就比较实惠了。如图6-10所示，这台盘阵机头带有一个扩展后端磁盘柜接口。

图6-10 带有一个扩展外部磁盘通道接口的控制器示意图

通道建好之后，下一步就是要扩充磁盘数量了。当然，JBOD就成了最佳选择。

图6-11所示的盘阵的每个控制器上多出一个额外的磁盘通道接口，这个接口露在机箱外面，用线缆连接了一个JBOD扩展柜。

图6-11 外接一个JBOD扩展柜的磁盘阵列

经过这样的改造，可连接的磁盘数量成倍增长。图中所示的是每个控制器增加了一个磁盘通道，还可以增加到两个或者多个通道。理论上，只要RAID控制器处理速度够强，总线带宽和面板上空间够大，多增加几个通道都没问题。

JBOD盘柜以前只有一个外部接口，为了配合双控制器，JBOD在其外部也增加了一个接口用来连接冗余的控制器。这样，扩展柜上也有两个外部接口了。

把带有控制器的磁盘柜称作“机头”，因为它就像火车头一样，是提供动力的。机头里可以有磁盘，也可以根本不含磁盘。把用于扩展容量用的JBOD叫做“扩展柜”，它就像一节节火车车厢，本身没有动力，全靠车头带，但是基本的供电和冷却系统还是要有的。图6-12所示的是IBM的DS400盘阵机头后视图，每控制器提供3个通道，机头内部的磁盘占用一个，然后另外两个提供扩展，在后面板上给出两个SCSI接口。图中Expansion ports所示的就是这两个SCSI接口。右边空白的地方是用来接入另外一个控制器的，这个控制器是可选组件。

图6-12 DS400盘阵的机头后视图

图6-13是用于连接DS400机头的扩展柜EXP400。可以看到它的左右各有一个接口模块，每个模块上有一个SCSI接口用来连接机头。

图6-13 EXP400扩展柜的后视图

6.6 锦上添花——完整功能的模块化磁盘阵列

再后来，机头做的都比较漂亮，而且感觉很厚实。但是JBOD就是一堆磁盘，显得和机头有些不搭配。所以也给扩展柜增加了所谓的模块，不仔细看的话，外观和机头没多大区别。只不过扩展柜的模块上，没有RAID控制器的功能，但是会加上一些其他功能，如探测磁盘温度等二线辅助功能。这个模块将接口、功能芯片、电路等都集成在一个板子上，所以外观和机头差不多。

图6-14所示的ESM模块，就是实现这些功能的插板。图6-15中所示的是一个磁盘扩展柜的实物后视图，可以看到上下两个模块，这两个模块不但负责链路通信，还负责收集设备各处的传感器发来的信息。

图6-14 一个扩展柜的前视图

图6-15 一个磁盘扩展柜的后视图

图6-14为一台盘阵的前视图。图6-15是一台FC接口的扩展柜后视图，可以看到上下两个ESH（Electrical Switch Hub）模块。这些磁盘扩展柜上的模块中主要包含单片机或者DSP芯片、FC-AL半交换逻辑处理以及其他功能的FPGA/ASIC/CPLD芯片、SFP适配器编码芯片、ROM或者Flash芯片（存放Firmware）、RAM缓存芯片（用于存放芯片执行程序时所需的数据）等，视设计不同而定。如果有新的Firmware被开发出来，可以将程序逻辑写入Flash或ROM芯片中，这个过程就是固件升级。FPGA/CPLD等芯片需要用外置的编程器写入新的电路逻辑。ASIC芯片不可升级，是固定逻辑的芯片，适用于成熟的、量产的芯片，比如SFP编码芯片等。

图6-16为扩展柜上的一个ESH2模块的内部实物图。

图6-16 ESH2模块实物图

6.7 一脉相承——主机和磁盘阵列本是一家

1. 盘阵（磁盘阵列）控制器的主机化

随着人们需求的不断提高，一个存储系统拥有几TB甚至几十几百TB的容量已经不是什么惊人的事情了。面对如此大的容量和如此多的磁盘，小小的控制器已经不能满足要求了。因此大的主机系统替代了短小精悍的控制器。

思考：可能有人已经不知所措了，用主机系统替代盘阵控制器，这不矛盾么？盘阵是给主机服务的，主机替代了盘阵控制器，岂不是乱了辈分了？

事实并非如此。众所周知，主机系统的经典架构就是CPU、内存、总线、各种IO设备和CPU执行的代码（软件），而观察一下盘阵控制器的基本架构，如RAID控制器芯片（CPU）、内存、总线、IO接口（SCSI接口等）和RAID芯片执行的代码（软件），就可以发现盘阵控制器就是一个简单的主机系统。

既然如此，完全可以用一台主机服务器来充当存储系统的控制器。比如，在这台主机上插入几张SCSI卡作为前端接口卡，再插入若干SCSI卡作为后端连接磁盘箱的接口卡，然后设计软件从/向后端读写数据，经过处理或者虚拟化之后，再传送给前端的主机服务器。

目前有两种趋势：一种是趋向使用现成的主机来充当控制器的载体，另一种是趋向使用高集成度的芯片作为控制器的核心。两种趋势各有利弊。

图6-17所示的是一台主机化的磁盘阵列实物图。

图6-17 主机化的盘阵控制器

2. 盘阵的类型

按照前端和后端接口来分，有SCSI-FC盘阵、FC-FC盘阵、SATA-FC盘阵、SCSI-SCSI盘阵等类型。SCSI-FC类型表示后端接口为SCSI接口，前端用于连接主机的为FC接口，也就是后端为SCSI磁盘，前端为FC接口的盘阵。

我们在后面会讲到FC-FC盘阵，这也是目前最高端的盘阵所采用的架构。图6-18所示的就是一台大型FC磁盘阵列的透视图，图示中一共5个机柜，中间的机柜整柜都为控制器，上方可见一排IO插卡，插卡上方为9个风扇。其余机柜中均为磁盘扩展柜。

图6-18 EMC DMX系列盘阵透视图

6.8 天罗地网——SAN

存储区域网络

大家来看最后一张图片，如图6-19所示。我们一开始描绘的那张“网中有网”的图片，现在大家应该能更深刻地理解了。网络，不仅仅指以太网、TCP/IP网，可以是SCSI网、PCI总线网、USB网等。RAID控制器，就相当于一个路由器，也就是协议转换器。

图6-19 网中有网

将磁盘放到了主机外部，存储设备和主机之间，就形成了又一个独立的网络：存储区域网络（Storage Area Network，SAN）。

数据就是在这种网络中来回穿梭，格式不断被转换和还原。

第7章 熟读宝典——系统与系统之间的语言**OSI**

■ OSI

■ 三元素

■ 七层结构

千百年来，江湖中人一直都把祖宗流传下来的一本宝典铭记于心，这本宝典就是号称“号令武林，莫敢不从”的OSI大典！

任何系统之间，如果需要通信，就有一套自己的协议系统。这个协议系统不仅要定义双方互相通信所使用的语言，还要规定所使用的硬件，比如通信线路等。例如以太网协议，凡是接入以太网的（交换机或者HUB）节点，都必须遵循以太网所规定的通信规程。两个对讲机之间进行通话，必须预先定义好发送和接收的频率，而且还要指定通信的逻辑，比如每说一句话之后，要说一个“完毕”，表示本地已经说完，该对方说了。

7.1 人类模型与计算机模型的对比剖析

人和计算机，二者有着天然的相似性。

7.1.1 人类模型

人类自身用语言来交流信息，这本身就是系统间通信的极佳例子。每个人都是一个系统，这个系统由八大子系统组成，每个子系统行使自己的功能，各个子系统相互配合，使得人体可以做出各种各样的事情，包括制造计算机系统这个“部分仿生”产物。

1. 消化系统

消耗系统负责食物的摄取和消化，使我们获得糖类、脂肪、蛋白质、维生素等营养，再经过一系列生化酶促反应，最终可以生成能量，以ATP分子的形式向各个细胞供应化学能，再经过一系列的分子机器（蛋白质或者蛋白质复合体）的处理，可以形成机械能、热、光、电等各种能量形式。

2. 神经系统

神经系统负责处理外部信息和调控人类自身运动，使我们能对外界的刺激有很好的反应，包括学习等活动也是在神经系统控制下完成的。比如皮肤、耳朵、眼睛等接收的各种信号，都会经过神经网络传送到大脑进行处理并做出反映。

3. 呼吸系统

呼吸系统是气体交换的场所，可以使人体获得新鲜的氧气。人类制造、储存和利用能量的每个过程，都需要氧气的参与，氧气是很好的氧化剂，人类利用这种氧化剂，氧化摄取的糖、蛋白质等物质，产生能量，如果没有氧气，则人体一切的生命活动就会终止，包括心肌的收缩和扩张运动。

4. 循环系统

循环系统负责氧气、营养和体液以及各种信号控制分子的运输，废物和二氧化碳的排泄，以及免疫活动。人体的这些空间中，各种器官按照规则分布着，每个器官行使其功能，必须供给它们能量、水和氧气以及各种控制因子蛋白，遍布周身的密密麻麻的动脉、静脉和毛细血管，就是运输这些物质必需的通道。这些物质溶在血液中，从动脉流入器官，完成生命逻辑后，从静脉流入肾脏，过滤废物，排泄，干净的静脉血再流回心脏，然后进入动脉，经过肺部的时候，将肺部获取的氧气吸溶入血液，成为鲜红色的动脉血，然后再流入各个器官，供给必须物质。

5. 运动系统

运动系统负责身体的活动，使我们可以做出各种动作。骨骼和肌肉属于运动系统，骨骼虽然不直接运动，但是它能在肌肉的牵引下进行运动。控制肌肉运动的是神经信号，大脑发出神经控制信号（一系列的蛋白质或者离子流），肌肉组织收到之后，便会引发雪崩似的化学反应，包括肌肉细胞中的微管结构的不断重组和释放，用这种形式来改变细胞形状，从而造成肌肉收缩或者扩张。而微管的组装和释放，需要能量和酶的参与，这时候，细胞便会贪婪地利用ATP分子来获取化学能，而血

液ATP损耗之后，血液便会加速流动到肌肉周围，提供更多的能量。这也是运动时心脏加速跳动的原因。而心脏跳动速度的控制因素很多，比如受到惊吓的时候，大脑便会分泌一些物质，传送到心肌，引发反应，促使跳动速度加快。

6. 内分泌系统

内分泌系统调节生理活动，使各个器官组织协调运作。内分泌系统是一个中央调控系统，比如生长素就是内分泌系统分泌的一种蛋白质分子，它被骨骼吸收后，可以促进骨骼生长。内分泌系统如果发生功能故障，则人体就会表现异常，包括精神和物理上的异常。

7. 生殖系统

生殖系统负责生殖活动，维持第二性征。生殖系统是人类繁衍的关键，生殖系统将人类的所有特征融入精子和卵子，受精卵在母亲子宫中逐渐发育成全功能的人体。

8. 泌尿系统

泌尿系统负责血液中生化废物的排泄，产生尿液。

7.1.2 计算机模型

我们发现，计算机系统和工作模式，与生物系统有很大一部分是类似的。

1. 计算机的消化系统

消化子系统是为整个系统提供基本能量和排泄消化废物的。给计算机提供能量的是电源和密布在电路板上的供电线路。为计算机排泄能量废物的，是接地低电位触点。电流流经供电线路，到接地触点终止，将能量提供给电路板上的各个部件（器官）。此外，计算机的“消化”也包含另外的意思，即吞入数据、吐出数据的过程。计算机从磁盘上读入数据，进行计算（消化）后，再输出数据。

2. 计算机的循环系统

循环子系统是为整个系统提供能量和物质传输通道的。给计算机提供数据传输通道的是各种总线，数据从总线流入CPU处理单元，完成计算逻辑后，又从总线输出到内存或者外设。所以我们说，计算机的循环系统，就是总线，心脏就是电路振荡装置。

3. 计算机的呼吸系统

呼吸子系统是为整个系统提供完成生命逻辑所必需的氧化剂，即氧气为计算机散热的风扇，貌似呼吸系统，但风扇的功能更类似于皮肤和毛孔的散热作用。

4. 计算机的神经系统

神经子系统的作用是传输各种信号，调节各个器官的功能。对于生物体来说，神经系统是运行在脑组织中的一种逻辑，这种逻辑通过执行一系列生化物理反应来体现它的存在。通过向血液中释放各种蛋白质信号分子，从而靶向各种器官，调节它们的功能。对于计算机系统来说，神经系统就是由CPU载入执行的程序，程序生成各种信号，通过总线传输给各个外部设备，从而调节它们的工作。

计算机的神经系统，可以认为就是外部硬件设备的驱动程序。神经网络就是控制总线，循环系统是数据总线。而生物体内没有地址总线的概念，血管凌乱的分布，并没有一种显式的区分机制来区分各个器官或者组，信号分子可以遍布周游全身。

5. 计算机的运动系统

计算机本身是一堆电路和芯片，不存在运动的概念。但是如果向计算机接口上接入了可以运动的部件，比如打印机、电动机、硬盘等，那么这些设备就可以在计算机控制信号（神经信号）的驱动下做运动，并且可以打印或者读写数据。

6. 计算机的生殖系统

目前，计算机系统表现的生殖能力，只是在一个硬件中生成不同的软件。软件通过CPU的执行，可以任意复制自身，并可以形成新的逻辑。在这种逻辑下，程序通过无数次的复制，难免在一些细微的Bug或者电路干扰的情况下发生奇特的变化，这些变化一开始可能不太会表现出来，但是随着量变的积累，就会引发质变，发生进化。

当然，计算机系统完全可以物理复制硬件，即通过程序控制外部机器，来生产新的计算机硬件，然后将软件复制到硬件上，继续繁殖。

7.1.3 个体间交流是群体进化的动力

人也好，计算机也好，他们之间都在不停地交流着。人和人的交流，让人类得到了进化。同样，计算机之间的交流，也会让计算机得到进化。交流是进化的动力，不可能有某种事物会完全脱离外界的刺激而

自身进化。

OSI便是这种交流所遵循的一张蓝图。

7.2 系统与系统之间的语言——OSI初步

OSI模型是一种被提取抽象出来的系统间通信模型。OSI中文的意思为“开放式系统互联”模型，是一个描述两个或者多个系统之间如何交流的通用模型。它不只适合于计算机系统互联，而且适合任何独立系统之间的互联。比如，人体和人体之间的通信，或者人体和计算机之间的通信，都可以用OSI模型来描述。

比如我和你之间需要交流，我们面对面坐着，此时我有一句话要和你讲：“您好，您怎么称呼？”

首先，我要说出这句话，要在脑海中生成这句话，即在语言处理单元中根据要表达的意思，生成符合语法的数据。然后通过神经将数据信号发送到声带、咬肌、舌头和口形固定之后，使声带振动。声带振动导致口腔空气共振，发出声音，经过空气机械波振动，到达你的耳膜接收器，耳膜被机械波谐振于一定频率，耳膜的振动通过神经信号传导到大脑，大脑相关的处理单元进行信号的解调，最终从神经信号中提取出我说的话“您好，您怎么称呼？”，这句话在大脑中，可能是离子流产生的模拟电信号，也可能是通过其他形式表示，比如一套编码系统。这句话被传送到大脑的语言处理单元，这个单元分析这句话，“您好”是一个问候语，当你知道我在问候你的时候，你的大脑会在这个“您好”信号的刺激下，进入一种“礼貌”逻辑运算过程，运算生成的信号，通过神经传送到你的颈部肌肉，收缩，使你的头部下降，并使面部肌肉收缩，这就完成了点头致意和微笑的动作。

这个过程，与计算机网络通信的模型完全一致。两台计算机之间通过以太网交换机相连，它们之间要进行通信。比如，a计算机想向b计算

机发送一个数据包，这个数据包的内容是“打开文件C:\tellme.txt”，过程如下所示。

(1) a计算机首先要在内存中通过双方定义的语言，生成这个数据包。

将这个数据包通过总线发送给TCP/IP协议处理单元，告诉TCP/IP处理单元对方的IP地址和所用的传输方式（UDP或TCP）和端口号。

TCP/IP处理模块收到这个包之后，将它包装好，通过总线发送给以太网卡。

以太网卡对数据包进行编码，然后通过电路将包装好的数据包变成一串电路的高低电平振荡，发送给交换机。

交换机将数据包交换到b计算机的接口。

(2) b计算机收到这串电位流后，将其输送到以太网卡的解码芯片，去掉以太网头，之后产生中断信号，将数据包送到内存。

由TCP/IP协议处理模块对这个数据包进行分析，提取IP头和TCP或UDP头，以便区分应输送到哪个应用程序的缓冲区内存。

最终TCP/IP协议将“打开文件C:\tellme.txt”这句话，成功输送到了b计算机应用程序的缓冲区内存中。

(3) b计算机应用程序提取这句话，分析它的语法，发现a计算机要求它打开C:\tellme.txt文件，则应用程序根据这个命令，调用操作系统打开文件的API执行这个操作。

分析一下上面的过程，我们发现如下内容。

数据总是由原始直接可读状态被转变成电路的电位振荡流，或者频率和振幅不断变化的机械波，也可能转换成一定频率的电磁波。

互相通信的两个系统之间必定要有连通的介质，空气、以太网或者其他形式，电磁波传递不需要介质。

相互通信的双方必须知道自己是在和谁通信。

以上三个要素，就是系统互联通信所具备的“连、找、发”三要素。

- 连：就是指通信的双方必须用某种形式连通起来，否则两个没有任何形式连通的系统之间是无法通信的。即便是电磁波通信，也至少通过了电磁波连通。
- 找：是说通信的双方或者多方，必须能够区分自己和对方以及多方（广播系统除外）。
- 发：定义了通信的双方如何将数据通过连通介质或者电磁波发送到对方。

7.3 OSI模型的七个层次

网络通信三元素抽象模型是对OSI模型的更高层次的抽象。OSI模型将系统间通信划分成了七个层次。

OSI模型的最上面的三层，可以归属到应用层之中，因为这三层都不关心如何将数据传送到对方，只关心如何组织和表达要传送的数据。

7.3.1 应用层

应用层是OSI模型的最上层，它表示一个系统要对另一个系统所传达的最终信息。比如“您好，您怎么称呼？”这句话，就是应用层的数据。应用层只关心应用层自身的逻辑，比如这句话应该用什么语法，应该加逗号还是句号？末尾是否要加一个问号？用“你”还是“您”等这样的逻辑。对于计算机系统来说，上文所述的例子中“open file C:\tellme.txt”，这条指令，就是应用层的数据。应用层程序不必关心这条指令是如何传达到对方的。

7.3.2 表示层

表示层就是对应用层数据的一种表示。如果前面说的“您好，您怎么称呼？”这句话是有一定附加属性的，例如“您好”这两个字要显示在对方的屏幕上，用红色显示在第1行的中央，而“您怎么称呼？”这几个字用蓝色显示在第10行的中央。这些关于颜色、位置等类似的信息，就构成了表示层的内容。

发送方必须用一种双方规定好的格式来表示这些信息，比如用一个特定长度和位置的字段来编码各种颜色（一般用三原色的组合编码来表

示），用一个字段来表示行列坐标位置。将这些附加表示层信息字段放置于要表达的内容的前面或后面，接受方按照规定的位置和编码来解析这些表示层信息，然后将颜色和位置信息赋予“您好，您怎么称呼？”这句话，显示于屏幕上。需要强调一点，表示层不一定非得是单独的一个结构体，它可以嵌入在实体数据中。这就是表示层，一些加密等操作就是在表示层来起作用的。

7.3.3 会话层

顾名思义，会话层的逻辑一定是建立某种会话交互机制。这种交互机制实际上是双方的应用程序之间在交互。它们通过交互一些信息，以便确定对方的应用程序处于良好的状态中。比如两个人通电话，拨通之后这个问：“能听清么？”那个说：“能听清，请讲。”这就是一个会话的过程。也就是说通信的双方在发送实际数据之前，先建立一个会话，互相打个招呼，以便确认双方的应用程序都处于正常状态。

应用层、表示层和会话层的数据内容被封装起来，然后交给了我们的货物押运员传输层。

TCP/IP协议体系模型中有4层，即应用层（应用访问层）、传输层、网络层和物理链路层（硬件访问层）。TCP/IP协议体系没有完全按照OSI匹配，它将OSI中的应用层、表示层和会话层统统合并为一层，叫做应用访问层，意思是指这个层全部是与应用程序相关的逻辑，与网络通信无关，应用程序只需调用下层的接口即可完成通信。

7.3.4 传输层

可以说OSI模型中上三层属于应用相关的，可以划入应用层范围，

而下四层就属于网络通信方面的。也就是说，下四层的作用是把上三层生成的数据成功地送到目的地。典型的传输层程序如下。

TCP协议的作用就是保障上层的数据能传输到目的地。TCP就像一个货运公司的押运员，客户给你的货物，就要保证给客户送到目的地，而不管你通过什么渠道，是直达（直连路由）还是绕道（下一跳路由），是飞机还是火车、轮船（物理线路类型）。

如果运输过程中出现错误，必须重新把货物发送出去。每件货物到了目的地，必须找收件人签字（TCP中的ACK应答包），或者一批货物到达后，收件人一次签收（滑动窗口）。

最后回公司登记。

提示： TCP还处理拥塞和流量控制。比如调度（路由器）选择了走这条路，但是太拥挤了，那么我也不好说什么，因为选哪条路到达目的是由调度（路由器）说了算，我只管押运。那么我只能通知后续的货物慢一点发货，因为这条路太挤了。当道路变得畅通时，我会通知后面的货物加速发货。这就是TCP的任务。TCP是通过接收方返回的ACK应答数据包来判断链路是否拥挤，比如发了一批货，半天都没接收到对方的签字，证明链路拥塞，有货物被丢弃了，那么就减缓发送速度。当有ACK被接收到后，我会增加一次发送货物的数量，直到再次拥塞。那么调度怎么知道这些货物是送到哪里的呢？这是网络层程序的任务。

注意： 传输层的程序一定要运行在通信双方的终端设备上，而不是运行在中间的网络互联设备上。传输层是一种端到端的保障机制，所谓端到端的保障就是指数据从一端发送到另一端

之后，对方必须在它的传输保障时间中成功收到并处理了数据，才能算发送成功。如果只是发送到了对方的网卡缓冲区，此时发生故障，如突然断电，这就不叫端到端的保障。因为数据在网卡缓冲区内，还没有被提交到TCP协议的处理逻辑中进行处理，所以不会返回成功信号给发送方，那么这个数据包就没有被发送成功，发送方会通过超时来感知到这个结果。

7.3.5 网络层

客户把货物交给货运公司的时候，必须填写目的地址（比如IP地址）。只要一个地址就够了，至于到这个地址应该坐几路公交车或哪趟火车等问题，客户统统不管，全部交给网络层处理。

货运公司为每件货物贴上一个地址标签（IP头）。

货运公司的调度们掌握了全球范围的地址信息（路由表），比如去某某地方应该走哪条路。

在选择了一条路之后，就让司机开车上路了。

押运员进行理货和收发货物，没事就在后车厢里睡觉。

此时最忙的是各个中转站的调度了。货物每次中转到一个地方就交给那个地方的调度，由那个调度来决定下一站应该到哪里。

接班的时候，旧调度不必告诉新调度最终目的应该怎么走，因为所有的调度都知道这个目的，一看就知道该走什么路了。

例如，有客户从新疆发货到青岛，由于新疆没有直达青岛的航班或者火车，所以只能先到达北京，然后再从北京直达青岛。

新疆的调度收到货物之后，他查找路由表，发现要到青岛，必须先到北京。新疆的调度会在货物上贴上青岛的标签而不是北京的标签，但是发货的时候，调度会选择将货物运送到新疆到北京的火车上。

货物到达北京之后，北京货运分公司的调度收到这件货物，首先查看这件货物的最终目的地址，然后北京调度也去查找路由表。他的路由表与新疆调度的路由表不同，在他的表上，北京到青岛有直达的火车，所以北京调度立即将货物原封不动的送上去青岛的火车。就这样一站一站的往前送（路由转发），货物最终从新疆到达了青岛。

思考： 那么调度是怎么知道全球地址表（路由表）的呢？这个表的生成是一个复杂的学习阶段，可以通过调度自行学习或者调度之间相互通告，也可以通过手工录入。前者称为动态路由，后者称为静态路由。

路由器充当的就是调度的角色。比如在青岛想访问一个位于北京的服务器，具体步骤如下。

首先必须知道这个服务器的IP地址，然后用这个IP地址作为最终目的地址组装成数据包，发送给位于青岛的Internet提供商机房中的路由器。

这个路由器收到这个包后，解析其目的IP地址，然后查找其路由表，发现这个目的IP地址的包应该从1号端口转发出去，所以它立即将这个包原封不动地向一号口转发。一号口通过光缆直接连接到了位于河北机房中的另一台路由器。

提示： 当然青岛到河北之间不可能只用一条连续不断的光缆连接，中途肯定经过一些光缆通信中继站。

河北的路由器收到这个IP包后，同样根据目的IP地址查找路由表，发现这个目的地址的包应该从8号端口中转发，它立即将这个包转发向8号端口。

8号端口通过光缆直接连接到了位于北京机房的一台路由器。

这台路由器同样查找路由表做转发动作。

经过一层层的寻找，最终找到了北京的这台服务器，将这个包传送到这台服务器的网卡，并提交到TCP/IP协议处理内存空间中。

经过解析和处理，服务器发现最终的数据是一个TCP握手数据包，然后TCP/IP程序立刻返回一个确认包，再次返回给服务器一个确认包。三次握手完成后，就可以向服务器发送HTTP请求来获取它的网页资源了。

7.3.6 数据链路层

数据链路层就是指连通两个设备之间的链路，数据要经过这条链路来传递给对方。数据链路层的程序将上层的数据包再次打包成对应链路的特定格式，按照对应链路的规则在链路上传输到对方。

数据链路就好比交通规则。在高速公路或者铁路上是需要遵守规则的，不能超速，不能乱停车，不能开车灯到最亮等。上路之前，先要看看公路的质量怎么样，是不是适合跑车或者先和对方商量一下传输的事宜。这就是链路层协商。

链路层的作用

首先是协商链路参数，比如双工、速率、链路质量等。

其次是将上层数据内容打包成帧，加上同步头进行传输，一次传输一句或者一个字符一个字符的传输（取决于上层的选择）。

最后，链路层程序调用物理层提供的接口，将帧提交给物理层。

相对于传输层的保障来说，OSI的数据链路层也提供一些保障机制。比如一些链路层协议会在每个帧后面加一个校验字段，如果对方收到的帧的校验值与这个校验字段不符，则证明链路受到干扰，数据产生畸变，那么就将这一帧视为无效帧直接丢弃，不会向上层报告这个错误，因为上层对链路层的错误不关心。而接收方的传输层协议会感知某个包没有到达或者不完整，接收方的传输层协议会要求发送方重新传送这个不完整或者没有接收到的包，也就是端到端的保障传输。链路层只侦错，不纠错，而传输层既侦错，又纠错。

根据OSI模型，两台路由器或者交换机之间传送数据也属于两个系统间的互联，那么它们也一定遵循OSI的模型。下面就来分析一下两台PC之间通信和两个路由器之间通信有什么区别。PC间通信我们上文已经描述过，下面来讲一下路由间的通信。

简单的路由器设备工作在OSI的第三层，即网络层。它只处理下三层的内容，只有下三层的处理逻辑，而没有上四层的处理逻辑。路由器收到包后，只检查包中的IP地址，不会改变任何IP头之上的其他内容，最简单的路由器甚至不会改变IP头。在一些带有诸如NAT功能的路由器上，可能会对IP包的源或者目的IP地址做修改。数据包流入路由器后，路由器只分析到第三层的IP头，便可以根据路由表完成转发逻辑。

如图7-1所示为通信路径上各个设备所作用的层次示意图，具体过

程如下。

图7-1 通信路径上各个设备所作用的层次

左边的PC机A连接到路由器A的以太网卡1上，路由器A的以太网卡2与路由器B的以太网卡2相连，右边的PC机B连接到路由器B的以太网卡1上。此时，要用PC机A上的IE浏览器访问位于PC机B上的Web服务，在IE浏览器的地址栏中输入PC机B的IP地址并按Enter键后，IE浏览器便会调用WinSock接口来访问操作系统内核的TCP/IP协议栈。IE浏览器告诉TCP/IP协议栈它所访问的目的IP地址和目的端口，并把要发送的数据告诉TCP/IP协议栈。IE浏览器发送给PC机B的数据，当然是一个HTTP GET请求，具体内容属于上三层，在这里不关心也不做分析。

TCP/IP协议栈收到这个数据之后，发现IE浏览器与PC机B当前并不存在连接，所以它首先要向PC机B上的TCP/IP协议栈发起连接请求，也就是TCP的三次握手过程。PC机A的TCP/IP协议栈先组装第一次握手IP包，组好后发送给操作系统内核缓冲区，内核调用网卡驱动程序从缓冲区内将这个IP包编码并在线路上传递出去。握手数据包很小，只要一个以太网帧就可以容纳。

这个帧最终到达路由器A的以太网卡1缓冲区内。以太网卡1产生中断信号，然后将这个帧去掉以太网头，发送到路由器A的内存中，等待IP转发逻辑模块的处理。运行在路由器A上的IP转发逻辑模块，其实就是IP路由协议计算模块，这个模块分析此IP包的头部目的IP地址，查找路由表以确定这个包将从哪个接口发送出去。IP路由运算一定要快速高效，才不至于对网络性能造成瓶颈。

路由器A查找路由表发现这个包应当从以太网卡2转发出去，所以

它立即将这个包发送到以太网卡2并通过线路传送到了路由器B的以太网卡2上。经过同样的过程，路由器B将这个包路由到PC机B的以太网卡缓冲区内，PC机B的网卡产生中断，将这个包通过总线传送到PC机B的TCP/IP协议栈缓冲区内存。

运行在PC机B上的TCP/IP协议栈程序分析这个包，发现IP是自己的，TCP端口号为80，握手标识位为二进制1，就知道这个连接是由源地址IP所在的设备向自己的80端口，也就是Web服务程序所监听的端口发起的握手连接。根据这个逻辑，TCP/IP协议栈返回握手确认IP包给PC机A，PC机A再返回一个最终确认包，这样就完成了TCP的三次握手。

握手成功后，PC机A上的TCP/IP协议栈立即在其缓冲区内将由IE浏览器发送过来的HTTP GET请求数据组装成TCP/IP数据包，发送给PC机B。PC机B得到这个数据包之后，分析其TCP端口号，并根据对应关系将数据放到监听这个端口的应用程序的缓冲区内存。

应用程序收到这个GET请求之后，便会触发Web服务逻辑流程，返回Web网页数据，同样经由PC机B的TCP/IP协议栈，发送给PC机A。

上述过程是一个正常通信的过程。

提示：如果在PC机B向PC机A传送网页数据的时候，路由器A和路由器B之间的链路发生了几秒钟的短暂故障后又恢复连通性，这期间丢失了很多数据。虽然这样，依靠TCP协议的纠错功能，数据依然会被顺序的传送给PC机A。

我们就来分析一下TCP是如何做到的。假如，在链路中断的时候，恰好有一个帧在链路上传送。发生故障后，这个帧就永久的丢失了。即使链路恢复后，路由器也不会重新传送这个帧。但PC机B由于很长时间

都没有收到PC机A的确认信息，便知道刚才发送的数据包可能已经被中途的网络设备丢弃了，所以PC机B上的TCP协议将重新发送这个数据包。

提示： 未接收到确认的包会存放在缓冲区内，不会删除，直到收到对方确认。

所以，即便中途经过的网络设备将这个包丢弃了，运行在通信路径最两端的TCP/IP协议，依然会重传这些丢弃的包，从而保障了数据传输，这也就是端到端的传输保障。只有端到端的保障，才是真正的保障，因为中途网络设备不会缓存发送的数据，更不会自动重传。

7.3.7 物理层

物理层的作用就是研究在一种介质上（或者真空）如何将数据编码发送给对方。如果选择公路来跑汽车，要根据沥青路或者土路来选用不同的轮胎；如果选择利用空气来跑飞机，则需根据不同的气流密度来调整飞行参数；如果选择了真空，则只能利用电磁波或者光来传输，可以根据障碍物等因素选择不同波长的波来承载信号；如果选择了海水，则要根据不同的浪高来调整航海参数。这些都是物理层所关心的。

物理层和链路层的区别

物理层和链路层是很容易混淆的两个层次。链路层是控制物理层的。物理层好比一个笨头笨脑的传送带，它不停地在运转，只要有东西放到传送带上就会被运输到对方。不管给它什么东西，它都一视同仁并且不会停下。

假设你我之间有一个不停运转的传送带，某时刻我有一大批货物要

传送给你，是否可以一股脑的把这些货物不停地放到传送带上，一下子传送给你呢？当然可以，但是那样将没有整理货物的时间，永远处于不停地从传送带上拿下货物的状态，货物越堆越多，最终造成崩溃。如果能将货物一批一批的传送过来，不但给予了双方充足的整理货物的时间，而且使得货物传输显得井井有条。而将货物分批这件事，传送带本身是不会做的，只能靠TCP或者IP来做。链路层给每批货物附加上一些标志性的头部或者尾部，接收方看到这些标志，就知道一批货物又来了，并做接收动作。

每种链路，都有自己的一个最适分批大小，叫做最大传输单元，MTU。每次传输，链路上最大只能传输MTU大小的货物。如果要在一次传输中传送大于这个大小的货物，超过了链路接收方的处理吞吐量，则可能造成接收方缓冲区溢出或者强行截断等错误。

TCP和IP这两个协议程序都会给货物分批。第一个分批的是TCP，下到IP这一层，又会根据链路层的分批大小来将TCP已经分批的货物再次分批，如果TCP分批小于链路层分批，则IP不需要再分。如果是大于链路层的分批，则IP会将货物分批成适合链路层分批的大小。被IP层分批的货物，最终会由接收方的IP层来再组装合并，但是由TCP分批的货物，接收方的TCP层不会合并，TCP可以任意分割货物进行发送而接收的时候并不做合并的动作。对货物的处理分析全部交由上层应用程序来处理，所以利用TCP/IP通信的应用程序必须对自己所发送的数据有定界措施。

说白了，物理层就是用什么样的线缆、什么样的接口、什么样的物理层编码方式，归零还是不归零，同步方式，外同步还是内同步，高电压范围，低电压范围，电气规范等的东西。

通过物理层编码后，我们的数据最终变成了一串比特流，通过电路振荡传输给对方。对方收到比特流后，提交给链路层程序，由程序处理，剥去链路层同步头、帧头帧尾、控制字符等，然后提交给网络层处理程序（TCP/IP协议栈等）。IP头是个标签，收件人通过IP头来查看这个货物是谁发的。TCP头在完成押运使命之后，还有一个作用就是确定由哪个上层应用程序来处理收到的包（用端口号来决定）。应用程序收到TCP提交的数据后，进行解析处理。

7.4 OSI与网络

网络就是由众多节点通过某种方式互相连通之后所进行的多点通信系统。既然涉及到节点与节点间的通信，那么就会符合OSI模型。

首先我们看看计算机总线网络。CPU、内存、外设三者通过总线互相连接起来，当然总线之间还有北桥和南桥，这两个芯片犹如IP网络中的IP路由器或者网桥。CPU与内存这两个部件都连接到北桥这个路由器上，然后北桥连接到南桥，南桥下连一个HUB总线，HUB上连接了众多的外设，这些外设共享这个HUB与南桥进行通信。

提示：说到HUB，不要认为是专指以太网中的HUB，HUB的意思就是一条总线。如果在这条总线上运行以太网协议，则就是以太网HUB；如果在这条总线上运行的是PCI协议，则就是PCI HUB（PCI总线）。

连接到以太网HUB上的各个节点，采用CSMA/CD的竞争机制来获取总线使用权，PCI总线同样采用仲裁竞争机制，只是实现方式不同。实现方式也可以称为协议，所以有以太网HUB和PCI HUB之分，也就是说HUB上运行的是不同的协议。当然以太网HUB设计要求远远比PCI HUB低，速度也低很多。

图7-2所示的模型是一个常见的小型网络，几台PC通过以太网HUB和路由器互相连接起来，然后通过运行在每台PC上的TCP/IP协议来通信。路由器的作用只是分析目的IP地址从而做转发动作。

图7-2 一个典型的网络

而我们再观察一下图7-3，发现除了连接各个组件之间的线路变成了并行多线路之外，其他没有什么大的变化。但是，这两个网络的通信过程是有区别的。上面的网络利用一种高级复杂的协议——TCP/IP协议来通信，而图7-3所示的网络是通过直接总线协议进行通信。在下面的网络中，各个部件之间的连线非常短，速度很高且非常稳定，自身就可以保障数据的稳定传输，所以不需要TCP这种传输保障协议的参与。在上面的网络中，各个部件之间可能相隔很远的距离，链路速度慢，稳定性不如主板上的导线高，所以必须运行一种端到端的传输保障协议，比如TCP协议，来保障端到端的数据传输。

图7-3 计算机总线网络

此外，上面的两个网络模型，其本质是相同的，因为它们两个都是从基本原理发展而来的。我们说，这两个网络模型都符合OSI这个抽象模型。再甚者，这两个模型都符合“连、找、发”抽象模型。

首先，所有部件之间都用了导线来连接。对于第一个模型，导线为双绞线或者其他形式的外部电缆；对于第二个模型，导线为电路板上印刷的蛇行线。这就是所谓“连”。

其次，这两个模型中都有寻址的逻辑。第一个模型利用IP地址作为寻址方法；第二个模型中利用地址总线作为寻址方法。这就是所谓“找”。

提示： 生物细胞之间的通信，同样符合OSI模型和“连找发”模型。细胞之间通过血管来传递信息，这就是“连”；通过配体—受体关系来找到目标，这就是“找”；血液流动将配体分子传递（广播）到人体的每个角落，这就是“发”。我国分子生态学创

始人向近敏曾经提出分子信息网络学说，就恰恰体现了网络的思想。在分子上层，还有细胞信息网络学说和遗传信息网络学说，它们一个比一个高层，一个比一个抽象。然而分子信息网络也不一定就是最底层的网络，或许还有原子信息网络、电子信息网络等。

最后，第一个模型利用TCP协议进行有保障的数据发送动作，第二个模型中由于线路非常稳定，不需要高级协议参与，而是直接利用电路逻辑从目标部件将数据复制过来。这就是所谓“发”。

网中有网

我们在以前的章节中，多次提到过“网中有网”这个词。而我们现在再来体会一下，发现计算机系统、计算机网络、Internet，这些系统，确实可以用网中有网来描述。计算机总线这个微型网络，通过一个网卡，接入以太网交换机或者 HUB，与其他计算机总线网络形成一个局域网，然后这个局域网再连接到路由器网关，从而连接到更大的网络，甚至Internet。

所有的网络，都按照OSI和“连找发”模型有条不紊地通信交互着，为我们服务。分子之间和细胞之间神奇地相互作用着，地球和月球有条不紊地旋转运行着，太阳系缓慢地自转，并围绕着更大的银河系旋转。

第8章 勇破难关——Fibre Channel协议详解

■ Fibre Channel

■ 网状通道协议

■ 光纤通道协议

■ OSI

本书的第6.8节，引出了SAN的概念。SAN首先是个网络，而不是指存储设备。当然，这个网络是专门用来给主机连接存储设备用的。这个网络中有着很多的元件，它们的作用都是为了让主机更好地访问存储设备。

SAN概念的出现，只是个开头而已，因为按照SCSI总线16个节点的限制，不可能接入很多的磁盘。要扩大这个SAN的规模，还有很长一段路要走。如果仅仅用并行SCSI总线，那么SAN只能像PCI总线一样作为主机的附属品，而不可能成为一个真正独立的“网络”。必须找到一种可寻址容量大、稳定性强、速度快、传输距离远的网络结构，从而连接控制器和磁盘或者连接控制器到主机。

干脆破釜沉舟，独立研发一套全新的网络传输系统，专门针对局部范围的高速高效传输。

然而，形成一套完整的网络系统并非易事，首先必须得有个蓝图。

这个蓝图是否有现成可以参考的呢？当然有，OSI就是一个经典的蓝图。OSI是对任何互联系统的抽象。

8.1 FC网络——极佳的候选角色

FC协议自从1988年出现以来，已经发展成为一项非常复杂、高速的网络技术。它最初并不是研究来作为一种存储网络技术的。最早版本的FC协议是一种为了包括IP数据网在内的多种目的而推出的高速骨干网技术，它是作为惠普、Sun和IBM等公司组成的R&D实验室中的一项研究项目开始出现的。曾经有几年，FC协议的开发者认为这项技术有一天会取代100BaseT以太网和FDDI网络。在20世纪90年代中期，还可以看到研究人员关于FC技术的论文。这些论文论述了FC协议作为一种高速骨干网络技术的优点和能力，而把存储作为不重要的应用放在了第二位。

Fibre Channel也就是“网状通道”的意思，简称FC。

提示： 由于Fiber和Fibre只有一字之差，所以产生了很多流传的误解。FC只代表Fibre Channel，而不是Fiber Channel，后者被翻译为“光纤通道”，甚至接口为FC的磁盘也被称为“光纤磁盘”，其实这些都是很滑稽的误解。

不过到目前为止，似乎称FC为光纤而不是直接称其FC的文章和资料更多。这种误解使得初入存储行业的人摸不着头脑，认为FC就是使用光纤的网络，甚至将FC与使用光纤传输的以太网链路混淆起来。在本书内不会使用“光纤通道”或者“光纤磁盘”这种定义，而统统使用FC和“FC磁盘”。相信在阅读完本章之后，大家就不会再混淆这些概念了，会知道FC与光纤根本就没有必然的联系。

Fibre Channel可以称为FC协议，或FC网络、FC互联。像TCP/IP一样，FC协议集同样具备TCP/IP协议集以及以太网中的很多概念，比如

FC交换、FC交换机、FC路由、FC路由器，SPF路由算法等。我们完全可以类比地看待TCP/IP协议以及FC协议，因为它们都遵循OSI的模型。任何互联系统都逃不过OSI模型，不可能存在某种不能归属于OSI中某个层次的元素。

下面我们用OSI来将FC协议进行断层分析。

8.1.1 物理层

OSI的第一层就是物理层。作为一种高速的网络传输技术，FC协议体系的物理层具有比较高的速度，从1Gb/s、2Gb/s、4Gb/s到当前的8Gb/s。作为高速网络的代表，其底层也使用了同步串行传输方式，而且为了保证传输过程中的电直流平衡、时钟恢复和纠错等特性，其传输编码方式采用N Mb编码方式。

为了实现远距离传输，传输介质起码要支持光纤。铜线也可以，但是距离受限制。FC协议集中物理层的电气子层名为FC0，编码子层名为FC1。

8.1.2 链路层

1. 字符编码以及FC帧结构

现代通信在链路层一般都是成帧的，也就是将上层发来的一定数量的位流打包加头尾传输。FC协议在链路层也是成帧的。既然需要成帧，那么一定要定义帧控制字符。

FC协议定义了一系列的帧控制策略及对应的字符。这些控制字符不是ASCII码字符集中定义的那些控制字符，而是单独定义了一套专门

用于FC协议的字符集，称为“有序集”。其中的每个控制字符其实是由4个8位字节组成的，称为一个“字”（word），而每个控制字开头的一个字节总是经过8 10b编码之后的0011111010（左旋）或者1100000101（右旋）。

由于还没有标准名词出现，所以不得不引入“左旋”和“右旋”这两个化学名词来描述这种镜像编码方式。左旋和右旋是指1和0对调。编码电路可以根据上一个10位中所包含的1的个数来选择下一个10位中1的个数。如果上一个1的个数比0的个数少，那么下一个10位中就编码成1的个数比0的个数多，这样总体平衡了1和0的个数。

0011111010左旋或者1100000101右旋，FC协议给这个字符起了一个名字，叫做K28.5。这个字未经过8 10b编码之前的值是十六进制BC，即10111100，它的低5位为11100（十进制的28），高3位为101（十进制的5）。FC协议便对这个字表示为“K28.5”，也就是说高三位的十进制是5，低5位的十进制是28，这样便可以组合成相应的二进制位码。然后再加上一个描述符号K（控制字符）或者D（数据字符）。K28.5这个字符没有ASCII字符编码与其冲突，它的二进制流中又包含了连续的5个1，非常容易被电路识别，当然符合这些条件的字符还有好几个。

每个控制字均由K28.5字符开头，后接3个其他字符（可以是数据字符），由这4个字符组成的字来代表一种意义，比如SOF（Start Of Frame）、EOF（End Of Frame）等。

定义了相关的控制字之后，需要定义一个帧头了。FC协议定义了一个24B的帧头。以太网帧头才14B，用起来还绰绰有余，为什么FC需要定义24B呢？在这个问题上，协议的设计者独树一帜，因为这24B的帧头不但包含了寻址功能，而且包含了传输保障的功能。网络层和传输

层的逻辑都用这24B的信息来传递。

我们知道，基于以太网的TCP/IP网络，它的开销一共是：14B（以太网帧头）+20B（IP头）+20B（TCP头）=54B，或者把TCP头换成8B的UDP头，一共是42B。这就注定了FC的开销比以太网加上TCP/IP的开销要小，而实现的功能都差不多。

可以看出，以太网中用于寻址的开销太大，一个以太网MAC头和一个IP头这两个就已经34B了，更别说再加上TCP头了。而FC将寻址、传输保障合并起来放到一个头中，长度才24B。图8-1所示的是一个FC帧的示意图，图8-2是一个FC帧编码之后的表示。

图8-1 一个FC帧的结构

图8-2 一个完整的FC帧的有序集表示

2. 链路层流量控制

在链路层上，FC定义了两种流控策略：一种为端到端的流控，另一种为缓存到缓存的流控。端到端流控比缓存到缓存流控要上层和高级。在一条链路的两端，首先面对链路的一个部件就是缓存。接收电路将一帧成功接收后，就放入了缓存中。如果由于上位程序处理缓慢而造成缓存已经充满，FC协议还有机制来通知发送方减缓发送。如果链路的一端是FC终端设备，另一端是FC交换机，则二者之间的缓存到缓存的流量控制只能控制这个FC终端到FC交换机之间的流量。

而通信的最终目标是网络上的另一个FC终端，这之间可能经历了多个FC交换机和多条链路。而如果数据流在另外一个FC终端之上发生拥塞，则这个FC终端就必须通知发起端降低发送频率，这就是“端到端”的流量控制。图8-3示出了这两种机制的不同之处。

图8-3 B2B和E2E两种方式的流量控制示意图

3. MTU

一般情况下，以太网的MTU为1500B，而FC链路层的MTU可以到2112B。这样，FC链路层相对以太网链路层的效率又提高了。

8.1.3 网络层

1. 拓扑

与以太网类似，FC也提供了两种网络拓扑模式：FC-AL和Fabric。

FC-ALFC-AL拓扑类似于以太网共享总线拓扑，但是连接方式不是总线，而是一条仲裁环路（Arbitral Loop）。每个FC AL设备首尾相接构成了一个环路。一个环路能接入的最多节点是128个，实际上是用了一个字节的寻址容量，但是只用到了这个字节经过810b编码之后奇偶平衡（0和1的个数相等）的值，也就是256个值中的134个值来寻址，这些被筛选出来的地址中又被广播地址、专用地址等占用了，最后只剩下127个实际可用的节点地址。

图8-4为4个FC-AL设备接入一个仲裁环的拓扑图。仲裁环是一个由所有设备“串联”形成的闭合环路。如果某个设备发生故障，这个串联的环路是不是就会全部瘫痪呢？在FC-AL集线设备的每个接口上都有一套“旁路电路”（Bypass Circuit），这套电路一旦检测到本地设备故障或电源断开，就会自动将这个接口短路，从而使得整个环路将这个故障的设备Bypass掉，不影响其他设备的工作。

图8-4 FC仲裁环结构示意图

数据帧在仲裁环内是一跳一跳被传输的，并且任何时刻数据帧只能按照一个方向向下游传输。图8-5为AL环路数据帧传输机制的示意图。

图8-5 AL环路数据帧传输机制示意图

在图8-5所示的仲裁环中，若a节点想与h节点通信，在a节点赢得仲裁之后，便向h节点发送数据帧。然而，由于这个环的数据是顺时针方向传递的，所以a发出的数据帧，只能先被b节点收到，由b节点接着传递到c节点，依次传递，最终传递到h节点。所以，虽然a和h节点之间只有一跳的距离，但是仍然需要绕一圈来传递数据。

Fabric另一种Fabric拓扑和以太网交换拓扑类似。Fabric的意思为“网状构造”，表明这种拓扑其实是一个网状交换矩阵。

交换矩阵的架构相对于仲裁环路来说，其转发效率大大提高了，联入这个矩阵的所有节点之间都可以同时进行点对点通信，加上包交换方式所带来的并发和资源充分利用的特性，使得交换架构获得的总带宽为所有端口带宽之和。而AL架构下，接入环路的节点不管有多少，其带宽总为恒定，即共享的环路带宽。

图8-6为一个交换矩阵的示意图。每个FC终端设备都接入了这个矩阵的端点，一个设备发给另一个设备的数据帧被交换矩阵收到后，矩阵便会“拨动”这张矩阵网交叉处的开关，以连通电路，传输数据。可以将这个矩阵想象成一个大的电路开关矩阵，矩阵根据通信的源和目的决定拨动哪些开关。这种矩阵被做成芯片集成到专门的交换机上，然后辅以实现FC逻辑的其他芯片或CPU、ROM，就形成了一台用于Fabric交换的交换机。

图8-6 Cross Bar交换矩阵示意图

图8-7所示的是一台Fabric交换机。FC设备通过光纤或者铜线等各种标注的线缆连接到这台交换机上，便可以实现各个节点基于FC Fabric拓扑方式的点对点通信。

图8-7 Brocade公司的FC交换机

FC交换拓扑寻址容量是2的24次方个地址，比以太网理论值（2⁴⁸）少。即便是这样，对于专用的存储网络来说也足够了，毕竟FC设计的初衷是用于存储网络的一种高速高效网络。

2. 寻址

任何网络都需要寻址机制，FC当然也不例外了。

首先，像以太网端口MAC地址一样，FC网络中的每个设备自身都有一个WWNN（World Wide Node Name），不管这个设备上有多少个FC端口，设备始终拥有一个固定的WWNN来代表它自身。然后，FC设备的每个端口都有一个WWPN（World Wide Port Name，世界范围的名字）地址，也就是说这个地址在世界范围内是唯一的，世界上没有两个接口地址是相同的。

FC Fabric拓扑在寻址和编址方面与以太网又有所不同。具体体现在以太网交换设备上的端口不需要有MAC地址，而FC交换机上的端口都有自己的WWPN地址。这是因为FC交换机要做的工作比以太网交换机多，许多智能和FC的逻辑都被集成在FC交换机上，而以太网的逻辑相对就简单了许多，因为上层逻辑都被交给诸如TCP/IP这样的上层协议实现了。然而FC的Fabric网中，FC交换机担当了很重要的角色，它需要处理到FC协议的最上层。每个FC终端设备除了和最终通信的目标有交互

之外，还需要和FC交换机打好交道。

WWNN每个FC设备都被赋予一个WWNN，这个WWNN一般被写入设备的ROM中不能改变，但是在某些条件下也可以通过运行在设备上的程序动态的改变。

WWPN和三个IDWWPN地址的长度是64位，比以太网的MAC地址还要长出16位。可见FC协议很有信心，认为FC会像以太网一样普及，全球会产生264个FC接口。然而，如果8B长度的地址用于高效路由的话，无疑是梦魇（IPv6地址长度为128b，但是鉴于Internet的庞大，也只好牺牲速度换容量了）。所以FC协议决定在WWPN之上再映射一层寻址机制，就是像MAC和IP的映射一样，给每个连接到FC网络中的接口分配一个Fabric ID，用这个ID而不是WWPN来嵌入链路帧中做路由。这个ID长24位，高8位被定义成Domain区分符，中8位被定义为Area区分符，低8位定义为PORT区分符。

这样，WWPN被映射到Fabric ID，一个Fabric ID所有24b又被分成Domain ID、Area ID、Port ID这三个亚寻址单元。

- **Domain ID:** 用来区分一个由众多交换机组成的大的FC网络中每个FC交换机本身。一个交换机上所有接口的Fabric ID都具有相同的高8位，即Domain ID。Domain ID同时也用来区分这个交换机本身，一个Fabric中的所有交换机拥有不同的Domain ID。一个多交换机组成的Fabric中，Domain ID是自动被主交换机分配给各个交换机的。根据WWNN号和一系列的选举帧的传送，WWNN最小者获胜成为主交换机，然后这个主交换机向所有其他交换机分配Domain ID，这个过程其实就是一系列的特殊帧的传送、解析和判断。

- **Area ID:** 用来区分同一台交换机上的不同端口组，比如1、2、3、4端口属于Area 1，5、6、7、8端口属于Area 2等。其实Area ID这一层亚寻址单元意义不是很大。我们知道，每个FC接口都会对应一块用来管理它的芯片，然而每个这样的芯片却可以管理多个FC端口。所以如果一片芯片可以管理1、2、3、4号FC端口，那么这个芯片就可以属于一个Area，这也是Area的物理解释。同样，在主机端的FC适配卡上，一般也都是用一块芯片来管理多个FC接口的。
- **Port ID:** 用来区分一个同Area中的不同Port。

经过这样的3段式寻址体系，可以区分一个大Fabric中的每个交换机、交换机中的每个端口组及每个端口组中的端口。

3. 寻址过程

1) 地址映射

既然定义了两套编址体系，那么一定要有映射机制，就像ARP协议一样。FC协议中地址映射步骤如下。

当一个接口连接到FC网络中时，如果是Fabric架构，那么这个接口会发起一个登录注册到Fabric网络的动作，也就是向目的Fabric ID地址FFFFFE发送一个登录帧，称为FLOGIN。

交换机收到地址为FFFFFE的帧之后，会动态地给这个接口分配一个24b的Fabric ID，并记录这个接口对应的WWPN，做好映射。

此后这个接口发出的帧中不会携带其WWPN，而是携带其被分配的ID作为源地址。

提示： 以太网是既携带MAC地址，又携带IP地址，在效率上打了折扣。

如果接口是连接到FC仲裁环网络中，那么整个环路上的节点会选出一个临时节点（根据WWPN号的数值，最小的优先级最高），然后由这个节点发送一系列的初始化帧，给每个节点分配环路ID。

提示： FC网络中的FCID都是动态的，每个设备每次登录到Fabric所获得的ID可能不一样。同样，FC交换机维护的Fabric ID与WWPN的映射也是动态的。

图8-8所示的是FC设备登录到Fabric过程示意图。

图8-8 FC设备登录Fabric网络的过程

2) 寻址机制

编址之后就要寻址，寻址则牵扯到路由的概念。

一个大的FC网络中，一般有多台交换机相互连接，它们可以链式级联，也可以两两连接，甚至任意连接，就像IP网络中的路由器连接一样，但是FC网络不需要太多的人工介入。如果将几台交换机连接成一个FC网络，则它们会自动地协商自己的Domain ID，这个过程是通过选举出一个WWPN号最小的交换机来充当主交换机，由主交换机来向下给每个交换机分配Domain ID，以确保不会冲突。

对于寻址过程，这些交换机会运行相应的路由协议。最广泛使用的路由协议就是SPF（最短路径优先）协议，是一种很健壮的路由协议。比如用于IP网络中的OSPF协议，FC网络也应用了这种协议。这样就可以寻址各个节点，进行各个节点无障碍地通信。

IP网络需要很强的人为介入性，比如给每个节点配置IP地址，给每个路由器配置路由信息及IP地址等，这样出错率会很高。FC网络中自动分配和管理各种地址，避免了人算带来的错误。FC采用自动分配地址的策略，一个最根本的原因是FC从一开始就被设计为一个专用、高效、高速的网络，而不是给Internet用的，所以自动分配地址当然适合它。如果给Internet也自动分配地址，那么后果不堪设想。

既然要与目的节点通信，怎么知道要通信的目标地址是多少呢？我们知道，FC被设计为一个专用网络，一个小范围、高效、高速、简易配置的网络。所以使用它的时候也非常简便，就像在Windows中浏览网上邻居一样。

每个节点在登录到FC网络并且被分配ID之后，会进行一个名称注册过程，也就是接口上的设备会向一个特定的目的ID发一系列的注册帧，来注册自己。这个ID实际上并没有物理设备与其对应，只是运行在交换机上的一套名称服务程序而已，而对于终端FC设备来说，会认为自己是在和一个真实的FC设备通信。对于Windows系统来说，每台机器启动之后，如果设置了WINS服务器，会向WINS服务器来注册自己的主机名和IP地址。

每台机器都这么做，所以网络中的WINS服务器就会掌握网络中的所有机器的主机和IP。同样，FC交换机上运行的这个名称服务程序，就相当于WINS服务器。但是其地址是唯一的、特定的，不像WINS服务器可以被配置为任何IP地址。也就是说在FC协议中，这个地址是大家都公认不会去改变的，每个节点都知道这个地址，所以都能找到名称服务器。其实不是物理的服务器，只是运行在FC交换机上的程序，也可以认为FC交换机本身就是这台服务器。

节点注册到名称服务之后，服务便会将网络上存在的其他节点信息告诉这个接口上所连接的设备，就像浏览网上邻居一样，所以这个接口上的设备便知道了网络上的所有节点和资源。

ZONE为了安全性考虑，可以进行人为配置，让名称服务器只告诉某个设备特定的节点。比如网络上存在a、b、c、d四个节点，可以让名称服务只向a通告b、c两个节点的存在，而隐藏d节点，这样a看不到d。但是这样做有时候会显得很保险，因为a虽然没有通过名称服务得到d的ID，但是如果将节点d的ID直接告诉节点a的话，那么它就可以和d主动发起通信。而这一切，交换机不做干涉，因为交换机傻傻的认为只要名称服务器没有向a通告d的ID，a就不会和d发起通信。

发生这种结果的原因是在物理上节点a和节点d并没有被分开，a和d总有办法通信。就像有时网上邻居里看不到一台机器，但是它明明在线，那么如果此时知道那台机器的地址，照样可以不通过网上邻居，直接和它通信。如果两个节点被物理隔开了，那么就真的无能为力了。前者实现隔离的方法叫做软ZONE，后者的做法叫做硬ZONE。

所谓ZONE，即分区的意思，同一个分区内的节点之间可以相互通信，不同分区之间的节点无法通信。软ZONE假设大家都是守法公民，名称服务器没有通告的ID就不去连接；而硬ZONE不管是否守法都会从底层硬件上强制隔离，即使某个节点知道了另外分区中某个节点的ID，也无法和对方建立通信，因为底层已经被阻断了。图8-9是一个Fabric ZONE的示意图。

图8-9 一个具有三个ZONE的Fabric

与目标通信从名称服务器得知网络上的节点ID之后，如果想发起和

一个节点的通信，那么这个设备需要直接向目的端口发起一个N_PORT Login过程来交换一系列的参数，然后再进行Process Login过程（类似于TCP向特定应用端口发送握手包一样），即进行应用程序间的通信。比如，FC可以承载SCSI协议和IP协议，那么Initiator端就需要向Target端对应的功能发起请求，比如请求FCP类型的Process Login，那么Target端就知道这个连接是用于FCP流量的。这些Login过程其实就是上三层的内容，属于会话层，和网络传输已经没什么关联了。这些Login的帧也必须经过FC下四层来封装并传输到目的地，就像TCP握手过程一样。

名称服务器只是FC提供的所有服务中的一个，其他还有时间服务、别名服务等，这些地址都是事先定死的。

Fabric网络中还有一种FC Control Service，如果节点向这个服务注册，也就是向地址FFFFFD发送一个State Change Registration（SCR）帧，那么一旦整个Fabric有什么变动，比如一个节点离线了，或者一个节点上线了、或者一个ZONE被创建了等，Fabric便会将这些事件封装到Registered State Change Notification（RSCN）帧里发送给注册了这项服务的所有节点。这个动作就像预订新闻一样，通常一旦节点被通知有这些事件发生之后，节点需要重新进行名称注册，以便从名称服务器得到网络上的最新资源情况，也就是刷新一下。

这些众所周知的服务都是运行在交换机内部的，而不是物理上的一台服务器。当然如果愿意的话，也完全可以用物理服务器来实现，不过这样做的话，在增加了扩展性的同时也增加了Fabric的操作难度。

以上描述的都是基于FC交换架构的网络，即Fabric（FC交换网络）。对于FC仲裁环架构的网络没有名称注册过程，环上的每个节点都对环上其他节点了如指掌，可以对任何节点发起通信。

提示： 有些机制可以把环路和交换结构融合起来，比如形成 Private loop、Public loop等，这方面会在下文中介绍。

FC的链路层和网络层被合并成一层，统称FC2。

8.1.4 传输层

FC的传输层同样也与TCP类似，也对上层的数据流进行Segment，而且还要区分上层程序，TCP是利用端口号来区分，FC则是利用 Exchange ID来区分。每个Exchange（上层程序）发过来的数据包，被FC传输层分割成Information Unit，也就相当于TCP分割成的Segment。然后FC传输层将这些Unit提交给FC的下层进行传输。下层将每个segment当作一个Sequence，并给予一个Sequence ID，然后将这个Sequence再次分割成FC所适应的帧，给每个帧赋予一个Sequence Count，这样便可以保证帧的排列顺序。接收方接收到帧之后，会组合成Sequence，然后根据Sequence ID来顺序提交给上层协议处理。图8-10显示了这种层次结构。图8-11为FC网络上的数据帧传输示意图。

图8-10 FC协议的层次结构

图8-11 Fabric网络上的帧

传输层还有一个重要角色，就是适配上层协议，比如IP可以通过FC进行传输，SCSI指令可以通过FC来传输等。FC会提供适配上层协议的接口，就是IP over FC及SCSI over FC。这里，FC只是给IP和SCSI提供了一种通路，一种传输手段，就像IP over Ethernet和IP over ATM一样。

FC也是通过发送ACK帧来向对方发送确认信息的，这个和TCP的实现思想一样。只不过一个ACK帧是24B加上CRC、SOF、EOF，一共

36B，而TCP的ACK帧为 $14+20+20=54\text{B}$ 。两者差别已经很明显了，两个帧看不出来，但是发送多了，差别就看出来了。要看累积效应。当然这么算是很粗略的，还需要包括进链路控制，帧间隙开销等。

在传输层上，FC定义了几种服务类型，也就是类似TCP/IP协议中规定的TCP、UDP。FC协议中的Class 1服务类型是一种面向连接的服务，即类似电路交换的模式，为通信的双方保留一条虚电路，以进行可靠的传输。Class 2类型提供的是一种带端到端确认的保障传输的服务，也就是类似TCP。Class 3类型不提供确认，类似UDP。Class 4类型是在一条连接上保留一定的带宽资源给上层应用，而不是像Class 1类型那样保留整个连接，类似RSVP服务。使用什么服务类型，会在端口之间进行PLogin的时候协商确定。

FC传输层被定义为FC4。

8.1.5 上三层

FC协议的上三层表现为各种Login过程、包括名称服务等在内的各种服务等，这些都是与网络传输无关的，但是的确属于FC协议体系之内的，所以这些内容都属于FC协议的上三层。

8.1.6 小结

综上所述，FC是一个高速高效、配置简单，不需要太多人为介入的网络。基于这个原则，为了进一步提高FC网络的速度和效率，在FC终端设备上，FC协议的大部分逻辑被直接做到一块独立的硬件卡片当中，而不是运行在操作系统中。如果将部分协议逻辑置于主机上运行，会占用主机CPU内存资源。

TCP/IP就是一种运行于主机操作系统上的网络协议，其IP和TCP或者UDP模块是运行在操作系统上的，只有以太网逻辑是运行在以太网卡芯片中的，CPU从以太网卡接收到的数据是携带有IP头部及TCP/UDP头部的，需要运行在CPU中的TCP/IP协议代码来进一步处理这些头部，才能生成最终的应用程序需要的数据。

而FC协议的物理层到传输层的逻辑，大部分运行在FC适配卡的芯片中，只有小部分关于上层API的逻辑运行于操作系统FC卡驱动程序中，这样就使FC协议的速度和效率都较TCP/IP协议高。这么做，成本无疑会增加，但是网络本来就不是为大众设计的，增加成本来提高速度和效率也是值得的。

8.2 FC协议中的七种端口类型

在FC网络中，存在七种类型的接口，其中N、L和NL端口被用于终端节点，F、FL、E和G端口在交换机中实现。

8.2.1 N端口和F端口

N端口和F端口专用于Fabric交换架构中。连入FC交换机的终端节点的端口为N端口，对应的交换机上的端口为F端口。N代表Node，F代表Fabric。用N端口模式连入F端口之后，网络中的N节点之间就可以互相进行点对点通信了。图8-12所示的是N端口和F端口的示意图。

图8-12 N端口和F端口

8.2.2 L端口

L端口指仲裁环上各个节点的端口类型（LOOP）。环路上的所有设备可以通过一个FCAL的集线器相连，以使得布线方便，故障排除容易。当然，也可以使用最原始的方法，就是首尾相接。图8-13所示的是利用集线器连接的拓扑。

图8-13 基于FCAL集线器的FCAL环路连接

1. 私有环

私有环，就是说这个FC仲裁环是封闭的，只能在这个环中所包含的节点之间相互通信，而不能和环外的任何节点通信。

2. 开放环

这个环是开放的，环内节点不但可以和环内的节点通信，而且也可以和环外的节点通信。也就是说可以把这个环作为一个单元连接到FC交换机上，从而使得环内的节点可以和位于FC交换机上的其他N节点通信。如果将多个开放环连接到交换机，那么这几个开放环之间也可以相互通信。

要实现开放环架构，需要特殊的端口，即下面描述的NL和FL端口。

8.2.3 NL端口和FL端口

NL端口是开放环中的一类端口，它具有N端口和L端口的双重能力。换言之，NL端口支持交换式光纤网登录和环仲裁。而FL端口是FC交换机上用于连接开放仲裁环结构的中介端口。

开放环内可以同时存在NL节点和L节点，而只有NL节点才能和环外的、位于FC交换结构中的多个N节点或者其他类型节点通信。NL节点也可以同时和L节点通信。图8-14为NL和FL端口示意图。

图8-14 NL和FL端口示意图

开放环的融合机制

FC-SW设备的工作方式是它会登录到网络（FLOGI），并在Name Server中注册（PLOGI）。设备要传输数据时会先到Name Server查询Target设备，然后到目标设备进行注册（PRLI），最后传输数据。

FC-AL的设备工作方式与此完全不同，在环路的初始化（LIP）过

程中，生成一个环路上所有设备地址的列表，被称作AL_PA，并存储在Loop中的每个设备上。当设备要与目标主机通信时，会到AL_PA中查询目标主机，然后根据地址进行通信。

要让一个私有环中的设备和Fabric中的设备达到相互通信，必须采用协议转换措施，因为FC AL和FC Fabric是两套不同的逻辑体系。

提示： 在本书第13章论述了关于“协议之间相互作用”即“协议杂交”方面的内容。如果阅读到那一章，再回头来研究，我们可以发现，NL端口和FL端口之间，完全就是一种Tunnel模式，它们利用FC AL的逻辑，承载FC Fabric的逻辑，也就是踩着AL走Fabric。比如Flogin、PLogin等这些帧，都通过AL链路来发向FL端口，而整个环中其他节点，对这个动作丝毫不知道，也不必知道。

如果采用MAP方式达到两种协议形式的最大程度的融合，也是完全可以的。下面描述的这种模式，就是采用了MAP的思想。

这种MAP的模式使环内的任何L节点可以和环外的任何N节点之间就像对方和自己是同类一样通信。也就是说环内的L节点看待环外的N节点就像是一个不折不扣的L节点。反过来，环外的N节点看待环内的L节点就像是一个N节点一样。这个功能是通过在交换机上的FL端口实现的，这个端口承接私有环和Fabric。在私有环一侧，它表现为L端口的所有逻辑行为，而对Fabric一侧，它则表现为N端口的行为，也就相当于一个N-L端口协议转换。这个接口可以把环外的N节点“带”到私有环内，同时把环内的节点“带”到环外。环内的L节点根本不会知道它们所看到的其实是环外的N节点通过这个特殊的L端口仿真而来的。

当然也要涉及到寻址的MAP，因为Fabric和AL的编址方式不同，所以需要维护一个地址映射，将环内的节点统统取一个环外的名字，也就是将L端口地址对应一个N端口地址，而这些地址都是虚拟的，不能和环外已经存在的N端口地址重合，这样才能让环外节点知道存在这么一些新加入Fabric的节点（其实是环内的L节点）。而要让环外节点知道这些新节点的存在，就要将这些新的节点注册到名称服务器上。因为Fabric架构中，每个节点都是通过查询名称服务器来获取当前Fabric中所存在的节点的。同样，要让环内的节点知道环外的N节点的存在，也必须给每个N节点取一个AL地址，让这些地址参与环的初始化，从而将这些地址加入到AL地址列表中。这样，环内的节点就能根据这个列表知道环内都有哪些节点了。

让各自都能看到对方，知道对方的存在，这只是完成了MAP的第一步。接下来，还要进行更加复杂的MAP，即协议交互逻辑的MAP。假如一个环内节点要和一个环外节点通信，这个环内节点会认为它所要通信的就是一个和它同类的L节点，所以它赢得环仲裁之后，会直接向这个虚拟AL地址发起通信。

这个虚拟AL地址对应的物理接口实际上是交换机上的仿真L端口，仿真L端口收到由环内节点发起的通信请求之后，便开始MAP动作。首先仿真L端口根据这个请求的目的地址，也就是那个虚拟地址，查找地址映射表，找到对应的N端口的Fabric地址。然后主动向这个N端口发起PLogin过程，也就是将AL的交互逻辑最终映射到了Fabric的交互逻辑。即AL向虚拟地址发起的通信请求，被仿真L端口MAP成了向真正的N端口发起PLogin请求，这就是协议交互逻辑的MAP。请求成功之后，仿真L端口便一边收集环内L节点发来的数据，一边将数据按照Fabric的逻辑转发给真正的N端口。反之亦然，N端口的逻辑，仿真L端口同样也会

MAP成AL环的逻辑。这样，不管是环外的N端口还是环内的L端口，它们都认为它们正在和自己的同类通信。

图8-15所示为开放环与Fabric融合的示意图。

图8-15 开放环融合机制示意图

同样是将环接入Fabric，开放环的扩展性就比私有环接入强。因为一个NL端口可以和环外的多个N端口通信。也就是说，NL端口和FL端口可以看成是隐藏在环中的N端口和F端口。它们如果要通信，不能像直连的N和F端口那样直接进行Fabric登录，而必须先突破环的限制，即先要赢得环仲裁，再按照交换架构的逻辑进行Fabric登录，接着N端口登录，然后进程登录。而这一切，环内其他节点不会感知到。

具有NL端口的设备既能和环内的L端口设备通信，又能和环外的N端口设备通信，同时具有N和L端口的逻辑，这一切都不需要仿真MAP，只需要一个Tunnel过程即可。而环内的L节点如果想与环外的N端口通信，由于L节点自身没有N端口的逻辑，必须经过FL端口的MAP过程。所以，称具有NL端口的设备为Public设备，即开放设备。而称具有L端口的设备为Private设备，即不开放的私有设备。

8.2.4 E端口

E端口是专门用于连接交换机和交换机的端口。因为交换机之间级联，需要在级联线路上承载一些控制信息，比如选举协议、路由协议等。

8.2.5 G端口

G端口比较特殊，它是“万能”端口，它可以转变为上面讲到过的任何一种端口类型，按照所连接对方的端口类型进行自动协商变成任何一种端口。

终端节点端口编址规则

各种终端节点端口（N、NL、L）的FC ID地址都是24b（3B）长。但是N端口只使用3B中的高2B，即高16b；L端口只使用3B中的低1B，即低8b；NL端口使用全部3B。没有被使用的字节值为0。

产生这种编址机制不同的原因，是3种端口的作用方式不同。L端口只在私有环内通信，而一个环的节点容量是128个，所以只用8b就可以表示了。N端口由于处于Fabric交换架构中，节点容量很大，所以用了16b表示，最大到65536个节点。而NL端口，因为既处于环中，又要和Fabric交换架构中的节点通信，所以它既使用N端口的编址，又使用L端口的编址，所以用了全部3B。图8-16为端口编址示意图。

图8-16 三种FC节点类型的编址异同

任何设备都可以接入FC网络从而与网络上的其他FC设备通信，网络中的设备可以是服务器、PC、磁盘阵列、磁带库等。然而，就像以太网要求设备上必须有以太网接口才能连入以太网一样，设备上必须有FC接口才可以连入FC网络。

8.3 FC适配器

想进入FC网络，没有眼睛和耳朵怎么行呢？FC网络的眼睛就是FC适配器，或者叫做FC主机总线适配器，即FC HBA（Host Bus Adapter）。值得说明的是，HBA是一个通用词，它不仅仅指代FC适配器，而可以指代任何一种设备，只要这个设备的作用是将一个外部功能接入主机总线。所以，PC上用的PCI/PCIE网卡、显卡、声卡和AGP显卡等都可以叫做HBA。

图8-17所示的就是PCI接口的FC适配器。

图8-17 FC适配卡

图8-18所示的是可以用来接入FC网络的各种线缆，可以看到SC光纤、DB9铜线和RJ-45/47线缆，它们都可以用于接入FC网络，只要对端设备也具有同样的接口。所以，千万不要认为FC就是光纤，这是非常滑稽的。

图8-18 各种接口的FC HBA

同样，也不要认为FC交换机就是插光纤的以太网交换机，这是个低级错误。称呼FC为光纤的习惯误导了不少人。FC协议是一套完全独立的网络协议，比以太网要复杂得多。FC其实是Fibre Channel的意思，由于Fibre和Fiber相似，再加上FC协议普遍都用光纤作为传输线缆而不用铜线，所以人们下意识的称FC为光纤通道协议而不是网状通道协议。但是要理解，FC其实是一套网络协议的称呼，FC协议和光纤或者铜线实际上没有必然联系。如果可能的话，也可以用无线、微波、红外线或紫外线等来实现FC协议的物理层。同样以太网协议与是否用光纤

或者铜线、双绞线来传输也没有必然联系。

所以“FC交换机就是插光纤的以太网交换机”的说法是错误的。同样“以太网就是双绞线”和“以太网就是水晶头”这些说法都是滑稽的。

FC适配器本身也是一个计算机系统，有自己的CPU和RAM以及ROM。ROM中存放Firmware，加电之后由其上的CPU载入运行。可以说它就是一个嵌入式设备，与RAID卡类似，只不过不像RAID卡一样需要那么多的RAM来作为数据缓存。

8.4 改造盘阵前端通路——SCSI迁移到FC

现在是考虑把原来基于并行SCSI总线的存储网络架构全面迁移到FC提供的这个新的网络架构的时候了！

但是FC协议只是定义了一套完整的网络传输体系，并没有定义诸如SCSI指令集这样可用于向磁盘存取数据的通用语言。而目前已经有了两种语言，一种是ATA语言（ATA指令集），另一种就是SCSI语言（SCSI指令集）。那么FC是否有必要再开发第三种语言？完全没有必要了。SCSI指令集无疑是一个高效的语言，FC只需要将SCSI语言拿来用就可以，但必须将这种语言承载于新的FC传输载体进行传送。

SCSI协议集是一套完整而不可分的协议体系，同样有OSI中的各个层次。物理层使用并行传输。SCSI协议集的应用层其实就是SCSI协议指令，这些指令带有强烈应用层语义。而我们要解决的就是如何将这些指令帧传送到对方。早期并行SCSI时代，就是用SCSI并行总线技术来传送指令，这个无疑是一个致命的限制。随着技术的发展，并行SCSI总线在速度和效率上已经远远无法满足要求。好在SCSI-3协议规范中，将SCSI指令语义部分（OSI上三层）和SCSI底层传输部分（OSI下四层）分割开了，使得SCSI指令集可以使用其他网络传输方式进行传输，而不仅仅限于并行SCSI总线了。

FC的出现就是为了取代SCSI协议集的底层传输模块，由FC协议的底层模块担当传输通道和手段，将SCSI协议集的上层内容传送到对方。可以说是SCSI协议集租用了FC协议，将自己的底层传输流程外包给了FC协议来做。

FC协议定义了FC4层上的针对SCSI指令集的特定接口，称为FCP，也就是SCSI over FC。由于是一个全新的尝试，所以FC协议决定先将连接主机和磁盘阵列的通路，从并行SCSI总线替换为串行传输的FC通路。而盘阵后端连接磁盘的接口，还是并行SCSI接口不变。

从图8-19中可以看到，连接主机的前端接口已经替换成了FC接口，原来连接在主机上的SCSI适配器也被替换成了FC适配器。

图8-19 前端FC、后端SCSI架构的盘阵示意图

经过这样改造后的盘阵，单台盘阵所能接入磁盘的容量并没有提升，也就是说后端性能和容量并没有提升，所提升的只是前端性能。因为FC的高效、高速和传输距离，远非并行SCSI可比。

理解： 虽然链路被替换成了FC，但是链路上所承载的应用层数据并没有变化，依然是SCSI指令集，和并行SCSI链路上承载的指令集一样，只不过换成FC协议及其底层链路和接口来传输这些指令以及数据而已。

从图8-19中可以看到，不管是主机上的FC适配器还是盘阵上的控制器，都没有抛弃SCSI指令集处理模块，被抛弃的只是SCSI并行总线传输模块。也就是抛弃了原来并行SCSI协议集位于OSI的下四层（用FC的下四层代替），保留了整个SCSI协议的上三层，也就是SCSI指令部分。

将磁盘阵列前端接口用FC替代之后，极大地提高了传输性能以及传输距离，原来低效率、低速度和短距离的缺点被彻底克服了。

8.5 引入FC之后

引入FC之后有如下优势。

1. 提高了扩展性

FC使存储网络的可扩展性大大提高。如图8-20所示，一台盘阵如果只提供一个FC前端接口，同样可以连接多台主机，办法是把它们都连接到一台FC交换机上。就像一台机器如果只有一块以太网卡，而没有以太交换机或HUB的话，那么只能和一台机器相连。如果有了以太网交换机或HUB，它就可以和N台机器连接。使用FC交换机的道理也一样，这就是引入包交换网络化所带来的飞跃。

图8-20 多主机共享盘阵

多台主机共享一台盘阵同时读写数据，这个功能在并行SCSI时代是想都不敢想的。虽然并行SCSI总线网络可以接入16个节点，比如15台主机和一台盘阵连入一条SCSI总线，这15台主机只能共享这条总线的带宽，假设带宽为320MB/s，如果15台主机同时读写，则理论上平均每台主机最多只能得到20MB/s的带宽。而这只是理论值，实际加上各种开销和随机IO的影响，估计每台主机能获得的吞吐量会不足10MB/s。再加上SCSI线缆最长不能超过25米，用一条宽线缆去连接十几台主机和盘阵的难度可想而知。

而引入FC包交换网络之后，首先是速度提升了一大截，其次由于其包交换的架构，可以很容易地实现多个节点向一个节点收发数据的目的。

2. 增加了传输距离

FC携带有现代通信的特质，比如可以使用光纤。而这就可以使主机和与远隔几百米甚至上千米（使用单模光缆）之外的盘阵相连并读写数据。

3. 解决了安全性问题

可能很多人还会有疑问，在图8-20所示的拓扑中，多个主机共用一台只有一个外部接口的盘阵不会冲突么？当然不会。第一，交换机允许多个端口访问同一个端口是一个分时复用的包交换过程，这个是毋庸置疑的。第二，盘阵上的FC前端接口允许多个其他端口进行Port Login过程。那么盘阵上的逻辑磁盘LUN可以同时被多个主机访问么？完全可以。SCSI指令集中有一个选项，即独占式访问或者共享式访问。

（1）独占式访问。

即只允许第一个访问某个目标节点的节点保持对这个目标节点的访问，第二个节点要向这个目标节点发起访问请求，则不被允许，除非上一个节点发出了释放指令。独占模式下，每台主机每次访问目标前都需要进行SCSI Reserve，使用完后再进行SCSI Release释放SCSI目标，这样其他节点才能访问那个目标。

（2）共享式访问，即允许任何人来访问，没有任何限制。

所以，盘阵上的任何LUN都可以被多台主机通过一个前端接口或者多个前端接口访问。这是一个优点，也是一个隐患。因为多个主机在没有相互协商和同步的情况下，一旦对同一个LUN都进行写操作的话，就会造成冲突。比如两台Windows主机正处于运行状态，它们都通过FC适

配卡识别到了磁盘阵列上的同一个LUN。此时主机A向这个LUN上写了一个文件，假设主机B已经将文件系统的元数据读入了内存，磁盘上的数据被主机A更改这个动作主机B是感受不到的。隔一段时间之后，主机B可能将文件系统缓存Flush到磁盘，此时可能会抹掉这个文件的元数据信息。

所以，在没有协商和同步机制的两台主机之间共享一个LUN是一件可怕的事情。要解决这个问题，可以每次只开一台机器，主机B想访问就必须把主机A关机或卸载该卷，然后主机B开机或挂载该卷，这样才能保证数据的一致性。但这样有点过于复杂。第二种办法就是使两台或者多台机器同时开机或同时挂载该卷，而让机器上的文件系统之间相互协商同步，配合运作。我写入的东西会让你知道。如果我正在写入，那么你不能读取，因为你可能读到过时的信息。

在文件系统上增加这种功能，需要对文件系统进行修改，或直接安装新的文件系统模块。这种新的文件系统叫做集群文件系统，能保证多个机器共享一个卷，不会产生破坏。

有些情况确实需要让两台机器同时可以访问同一个卷（如集群环境），但是大多数情况下是不需要共享同一个卷的，每台机器拥有各自的卷，都只能访问属于自己的卷，这样不就太平了么？

是的，要做到这一点有两种方法。分析从主机到盘阵上的LUN的通路，可以发现通路上有两个部件，第一个部件是FC网络交换设备，第二个部件就是磁盘阵列控制器。可以在这两个部件上做某种“隐藏”或者“欺骗”，让主机只能对属于它自己的LUN进行访问。

（3）在磁盘阵列控制器上做“手脚”。

SCSI指令集中有一条指令叫做Report LUN，也就是在SCSI发起端和目标端通信的时候，由发起端发出这条指令，目标端在接收到这条指令之后，就要向发起端报告自己的LUN信息。可以在这上面做些手脚，骗发起端一把。当发起端要求Report LUN的时候，盘阵控制器可以根据发起端的唯一身份（比如WWPN地址），提供相应的LUN报告给它。

比如针对主机A，控制器就报告给它LUN1、LUN2、LUN3。虽然盘阵上还配置很多其他的卷，比如LUN4、LUN5、LUN6等，但是如果告诉控制器，让它根据一张表8-1所示的映射表来判断应该报告给某个主机哪个或哪些LUN的话，控制器就会乖乖地按照指示来报告相应的LUN给相应的主机。

表8-1 LUN映射表

如果某个主机强行访问不属于它的LUN，盘阵控制器便会拒绝这个请求。上面那张映射表完全需要人为配置，因为盘阵控制器不会知道我们的具体需求。所以对于一个盘阵来说，要想实现对主机的LUN掩蔽，必须配置这张表。

盘阵上的这个功能叫做LUN Masking（LUN掩蔽），也就是对特定的主机报告特定的LUN。这样可以避免“越界”行为，也是让多台主机共享一个盘阵的方法，从而让多台主机和平共享一台盘阵资源。毕竟，对于容量动辄几TB甚至几百TB的大型盘阵来说，如果不加区分的让所有连接到这台盘阵的主机都可以访问到所有的卷是没有必要的，也是不安全的。

不仅FC接口盘阵有这个功能，SCSI前端接口盘阵照样可以实现这个功能，因为这是SCSI指令集的功能，而不是传输总线的功能。不管用

什么来传输SCSI指令集，只要上面能承载SCSI指令集，那么指令集中所有功能都可用。

磁盘阵列除了可以将某些LUN分配给某个主机之外，还可以配置选择性地将某个或某些LUN分配到某个前端端口上。也就是说，设置前端主机只有从某个盘阵端口进入才能访问到对应的LUN，从盘阵前端其他端口访问不到这些LUN。有些双控制器的盘阵可以定制策略将某些LUN分配到某个控制器的某些端口上。LUN Masking的策略非常灵活，只要有需求就没有开发不出来的功能。

总之，可以把LUN当作蛋糕，有很多食客（主机）想吃这些蛋糕。然而，食客要吃到蛋糕，需要首先通过迷宫（FC网络），然后到达一个城堡（磁盘阵列）。城堡有好几个门（盘阵的前端接口），如果城堡的主人很宽松，会把所有蛋糕分配到所有门中，从任何一个门进入都可以吃到所有蛋糕。如果主人决定严格一些，那么他也许会将一部分蛋糕分配到1号门，另一部分蛋糕分配到2号门。如果主人非常严格，那他会调查每个食客的身份，然后制定一个表格，根据不同身份来给食客不同的蛋糕。

（4）在FC交换设备上做“手脚”。

我们前面提到过ZONE。ZONE的功能就是在FC网络交换设备上阻断两个节点间的通路，这样某些节点就根本无法获取并访问到被阻断的其他节点，也就识别不到其上的LUN了。LUN masking只是不让看见某个节点上的某些LUN而已，而ZONE的做法更彻底，力度更大。

ZONE有软ZONE和硬ZONE之分。软ZONE就是在名称服务器上做手脚，欺骗进行名称注册的节点，根据ZONE配置的信息向登录节点通

告网络上的其他节点以及资源的信息。硬ZONE就是直接把交换机上某些端口归为一个ZONE，另一些端口归为另一个ZONE，在两个ZONE之间完全底层隔离，端口之间都不能通信，如图8-21所示。

图8-21 ZONE示意图

图中有两个ZONE，FC盘阵B所连接的交换机端口既在左边的ZONE中，又在右边的ZONE中，这样是允许的。这个例子中，主机C是无法和盘阵A通信的，它只能识别到盘阵B上的LUN。

有了LUN Masking和ZONE，FC网络的安全就得到了极大的保障，各个节点之间可以按照事先配置好的规则通信。

8.6 多路径访问目标

再来看一下图8-22。这是一个具有双控制器的盘阵，两个控制器都连接到了交换机上，而且每个主机上都有两块FC适配卡，也都连接到了交换机上。前文说过，如果在盘阵上没有做LUN Masking的策略，而在FC交换机上也没有做任何ZONE的策略，则任何节点都可以获取到网络上所有其他节点的信息。

图8-22 多路径访问示意图

假设盘阵上有一个LUN1被分配给控制器A，LUN2被分配给控制器B，那么可以计算出来，每个主机将识别到4块磁盘。因为每个主机有两块FC适配卡，每个适配卡又可以识别到控制器A上的LUN1和控制器B上的LUN2。也就是说，每台主机会识别到双份冗余的磁盘，而主机操作系统对这一切一无所知，它会认为识别到的每块磁盘都是物理上独立的，这样很容易造成混乱。

既然会造成混乱，那么为何要在一台主机上安装两块FC适配卡呢？这样做就是为了冗余，以防止单点故障。一旦某块FC卡出现了故障，另一块卡依然可以维持主机到盘阵的通路，数据流可以立即转向另外一块卡。

如何解决操作系统识别出多份磁盘这个问题呢？办法就是在操作系统上安装软件，这个软件识别并分析FC卡提交上来的LUN。如果是两个物理上相同的LUN，软件就向操作系统卷管理程序提交单份LUN。如果某块FC卡故障，只要主机上还有其他的FC卡可以维持到FC网络的通路，那么这个软件依然会向操作系统提交单份LUN。一旦所有FC卡全

都故障了，主机就彻底从FC网络断开了，这个软件也就无法提交LUN了，操作系统当然也识别不到盘阵上的LUN了。

此外，如果盘阵的某个控制器接口发生故障，主机同样可以通过这个软件立即重定向到另一个备份控制器，使用备份控制器继续访问盘阵。

这种软件叫做“多路径”软件，中高端产品的开发商都会提供自己适合不同操作系统的多路径软件。多路径软件除了可以做到冗余高可用性的作用之外，还可以做到负载均衡。因为主机上如果安装了多块FC适配卡，数据就可以通过其中任何一块卡到达目的，这样就分担了流量。

提示： 多个存储适配器可以以active/standby模式或者active/active模式以及dual/multi active模式工作。active/standby模式是指同一时刻只能有一个适配器在收发数据，active/active模式是指同一时刻多个适配器可以共同收发针对同一个LUN的数据。而dual/multi active则是两个或指多个适配器不能同时针对同一LUN收发数据，但是每个适配器可以针对不同的LUN收发数据。

多路径软件示例

如图8-23所示为EMC公司针对其存储产品所开发的多路径软件PowerPath的配置监控界面。可以看到这台Windows主机上安装了4块FC卡。存储系统向这台主机共映射了7个LUN，分别对应Disk 001~007。

图8-23 PowerPath界面

如图8-24所示，其中一个LUN存在16条不同的路径。

图8-24 每个LUN通过16条路径被访问

如图8-25所示，我们可以判断出整个系统的拓扑。

图8-25 系统拓扑图

如图8-26所示，一块FC卡出现故障后，系统界面会显示出来。如图8-27所示，虽然一块FC卡出现了故障，但是每个LUN也只是丢失了16条路径中的4条，存储访问依然正常。

图8-26 一块FC卡出现故障

图8-27 LUN依然可以通过剩余的12条路径被访问

多路径软件与阵列控制器配合切换过程简介

如图8-28所示为四种典型的连接拓扑下各种链路故障情况的示意图。RDAC（Redundant Disk Array Controllers）是Linux下的一个多路径软件驱动程序，我们就用它的作用行为来给大家做介绍。多路径软件一般位于适配器驱动程序之上，对适配器上报的多份重复的LUN进行虚拟，虚拟成一个单一的逻辑设备然后再次上报。在Windows下多路径软件属于一种过滤驱动程序层（Filter Driver）。

图8-28 四种典型拓扑下的多路径切换示意图

下面我们就来看看这些情况下多路径软件到底会怎么来动作。

（1）在第一个场景中，LUN1的Owner（或称Prefer）控制器为A，而LUN2的Owner控制器为B。主机从两条路径分别认识到了这两个LUN，1个LUN1和1个LUN2。多路径软件会从HBA1链路来访问LUN1，从HBA2链路来访问LUN2。某时刻HBA2连接交换机的链路发生故障，那

么此时对LUN1的访问路径不受影响，但是对LUN2的访问链路完全中断，此时多路径软件必须切换到HBA1的链路来同时承载LUN1与LUN2的流量。由于LUN2的Owner控制器为B，所以此时有两种办法可以继续保持对LUN2的访问：第一种办法就是主机将IO通过HBA1→交换机A传送给控制器A，然后控制器A将IO请求通过控制器间的缓存镜像链路转发到控制器B，控制器B执行完毕后将结果返回给控制器A，之后原路返回给主机；第二种做法则是多路径软件在感知到故障之后，判断出只能从HBA1的链路走到控制器A了，那么此时多路径软件可以向控制器A发送命令，让它强行接管对LUN2的控制权，接管之后，针对LUN2的IO就无需再转发给控制器B了，直接由控制器A全权处理。由于第一种方式需要耗费镜像通道的带宽，所以出于性能考虑，一般都会使用第二种方式处理，即切换Owner控制器。

（2）在第二个场景中，阵列的双控制器各通过一条链路连接到一个交换机上。此时主机端可以看到共4个LUN，从HBA1链路看到一个LUN1和一个LUN2，从HBA2链路看到一个LUN1和一个LUN2。某时刻HBA1链路故障，那么此时毫无疑问，多路径软件一定要切换到HBA2链路继续收发IO。那么阵列控制器之间是否需要切换LUN的控制权呢？不需要，因为主机此时可以从HBA2→交换机B来看到分别被控制器A与控制器B管控的LUN1与LUN2。

（3）在第三个场景中，有两台主机分别用两块HBA来连接交换机了。LUN1只映射给主机1，而LUN2只映射给主机2。某时刻，主机1的HBA1链路故障或者卡件/接口故障，同时，阵列B连接交换机B的链路也发生故障。此时，主机1一定要切换到HBA2路径，通过交换机B到控制器A从而保持对LUN1的访问。而主机2则根据之前的优选路径来判断是否切换，如果之前的优选路径是通过HBA2→交换机B→控制器B的

话，那么此时就需要切换到HBA1，走交换机A再到控制器B了。

（4）在第四个场景中，LUN1与LUN2的Owner控制器均属于控制器B，LUN1只映射给主机1，而LUN2只映射给主机2。此时主机2的HBA2链路发生故障，那么此时主机1不受影响，依然走HBA2→交换机B→控制器B的路径来访问LUN1；而主机2此时必须切换到HBA1来收发IO，但是HBA1到控制器B并没有直接路径，必须通过双控制器之间的镜像路径，而这个之前也说过，不推荐使用，虽然理论上是可以做到的。那么此时主机2别无他法，只能通过HBA1向阵列的控制器A发送命令，将自己所要IO的LUN2的Owner控制器切换到控制器A上，主机2并不会要求将LUN1也切换，因为主机2只能感知到自己所访问的LUN，也只会要求切换自己要访问的LUN。

（5）如图8-29所示的第五个场景中为另外一种情况，即阵列控制器整机故障的情况。此时另外一个控制器会通过之间的镜像通道（同时也充当心跳线）感知到对方阵列已死，那么本端就会强行将对端控制器之前所管控的所有LUN无条件接管。同时，主机端多路径软件也需要根据情况改变优选路径到控制器A而不是已死的控制器B了。

图8-29 阵列控制器整机故障场景

（6）第六个场景中，阵列控制器B连接本地磁盘扩展柜的链路故障，这样就导致控制器B认不到本地下挂的所有磁盘了，但是依然可以认到控制器A处的磁盘（控制器B有链路连接到控制器A下面的磁盘柜）。此时控制器B可以有两种做法：第一种则是将原本处于其下挂磁盘上的LUN2的Owner管控权交给控制器A，并且在其前端强行unmap掉LUN2，这样主机端的多路径软件就可以感知到LUN2的消失，自动切换到另一条路径走控制器A；第二种做法就是不让主机端多路径软件感知

到任何变化，主机针对LUN2的IO依然下发到控制器B，而控制器B接收到IO之后，将其通过双控之间的镜像路径转发到控制器A处理（控制器A依然可以访问到挂在控制器B后面的磁盘），然后控制器A将结果返回给控制器B，之后控制器B再返回给主机。一般情况下可以针对不同场景做出选择，多路径切换过程会影响主机侧应用程序，但是不切换的话，数据都走镜像通道，性能会有所下降。

8.7 FC交换网络节点4次Login过程简析

每个FC节点连到FC Fabric网络里需要经历4次Login过程。

第一次Login相当于TCPIP网络里的DHCP过程，FC交换机需要为每个FC节点分配一个Fabric ID，相当于IP地址，有了这个ID，数据包才能被FC交换机正确地交换，FC交换机是根据Fabric ID而不是WWPN（相当于以太网的MAC地址）作交换的。

第二次Login过程，相当于Windows里的WINS服务器注册和资源发现过程，我们熟知的网上邻居，有两种访问方式，一种是广播方式，另一种是所有Windows PC都向WINS服务器（其IP地址预先在每台PC上被配置好）注册，双击网上邻居时候每台PC都会从WINS服务器拉取目前网络上的PC信息。FC也有这个过程，FC节点在FC Fabric里的第二次Login过程，就是向Name Server注册自己，并拉取目前FC网络里的所有Target节点信息（只有FC Initiator节点才会主动拉取资源，Target节点只注册不拉取），在第二次Login的过程中，其实包含了两次“子Login”过程，每个FC节点要注册到Name Server，必须先向Name Server发起Port Login过程，Port Login其实是指FC网络底层端口级别的Login，一个Fabric ID所在的端口要与另一个Fabric ID所在的端口发起通信，必须先Port Login，成功之后，再发起Process Login，所谓“Process Login”就是进程级别的Login，就是发起端的程序要向对方表明我将与你处运行的哪个程序通信，这就相当于TCPIP的端口号，到底要连接对方的哪个端口，每个端口都有一个上层应用程序在监听，向Name Server注册，那么Name Server上一定要运行一个管理注册过程和资源列表的程序，发起端就是在声明要与这个程序连通，从而注册自己，所以要向对方的FC底层协议栈声明“请将数据包发送给注册和资源管理这个Process”，所

以才叫做“Process Login”，与TCPIP向某端口的三次握手机制类似。经过这两次子Login，发起端才真正地与Name Server上的程序进行数据交互，从而完成注册和资源拉取过程。

第三次Login过程，就是FC Initiator节点向所有自己看到的Target节点发起Port Login。

成功之后，就开始第四次Login，也就是向Target节点发起Process Login，这里的“Process”一定就是对方的FCP Target程序了，这个程序被集成在了FC卡的Port Driver的下层。

第9章 天翻地覆——FC协议的巨大力量

■ Fibre Channel

■ SCSI

■ 前端

■ 后端

■ 机头

■ 扩展柜

■ FC磁盘

话说FC协议横空出世，在江湖上引起了轩然大波，各门派纷纷邀请FC协议来参与存储磁盘阵列的制造，用FC协议实现盘阵与主机的连接。

以前并行SCSI的时代已经结束，终于可以将那又宽又短的电缆彻底抛弃，取而代之的是细长的光纤。

然而，FC的出现并没有终结这场革命，SAS的二次革命又要到来！

9.1 FC交换网络替代并行SCSI总线的必然性

历史是不断前进的，事物也是不断发展的，新技术必定取代旧技术。FC取代并行SCSI总线有两个根本原因。

9.1.1 面向连接与面向无连接

在并行SCSI总线时代并没有复杂的链路层协议，“连”就体现在线缆上，就像连接CPU和北桥之间的铜线一样，只不过SCSI线缆被做成了柔软的外置线缆而已。基本在这种短线缆上可以不必考虑通信层面的内容，因为距离很短，线路是稳定的，不需要加入诸如传输保障机制之类的东西。同时，这种情况也相当于面向连接的电路交换，通信的双方要预先建立一条物理上的通路（虚电路），不管有没有数据流，这条通路总是存在，且带宽固定，别人也抢不走这条电路的使用权，这就给通信双方提供了最大的质量和稳定性保证。在这样的链路上，不需要过多的底层传输协议开销。

相反，在面向无连接的包交换网络中，数据流被封装成数据包，传输保障和流量控制等因素就显得十分重要了。因为此时网络是共享的，网络按照Best Effort尽力而为地转发数据包。以太网和FC交换网络都属于这种面向无连接的技术（FC中Class1类型服务除外）。

而电话交换网、并行SCSI总线网就属于面向连接的网络。当你提机拨号的时候，电话局的电话交换机便会在你和你通信的对方之间建立起一条物理电路，从而使双方通信。

提示： 大家可以观察一下电话交换机，每当有外线拨入的时候，就可以听到交换机里有吧嗒吧嗒的声音，这就是交换机在

做继电器开合动作。

对于并行SCSI总线，当通信发起者需要和某个节点通信的时候，它会申请总线仲裁，在获得总线资源之后，便直接和对方发起通信，此时并没有一个显式的连接建立的过程。物理通路总是存在的，在任意两点之间都存在，只不过此通路是个总线，是大家共享的，需要通过仲裁来获得总线使用权，也就等价于建立独享连接。SCSI指令和数据可以直接在这个总线上传递，并不需要过多的额外的协议开销。

1. 面向连接的致命弱点

但是面向连接的通信有三个致命弱点。

面向连接网络的第一个弱点，就是资源浪费。特别是在交换环境中，由于不管路径上有没有数据传输，这条预先建立的连接必须保持并且只给特定的通信双方使用，其他节点的通信不管数据多么拥塞都不能使用这条路径。面向连接的网络好比一个城市的公交系统，每条公交线路都是固定的，不管这条线路上的客流量多少，就算没有人坐这条线路了，公交车也要来回跑。而面向无连接的网络就好比出租车。在没有人的时候，出租车可以等待客人到来。一旦有客人到来，出租车便会上路，而且路线不是固定的，司机可以按照目前道路流量情况，选择空闲的道路前往目的地。

面向连接网络的第二个弱点是维持和维护这条连接所耗费的成本高。通信双方距离近时，没什么问题，但是一旦距离很远，要维持这条物理连接，就需要很高的成本。要解决长距离传输的干扰问题、需要中继等，这也是长途电话费居高不下的一个原因。

面向连接网络的第三个弱点，就是缺乏高可用性。一旦建立好的虚电路因为某种原因断开了，就需要通信发起者重新建立电路才能继续双方的通信。这种现象在打长途电话的时候经常遇到，此时不得不重新拨打。而对于包交换网络，通信双方没有一条固定的数据流路径，交换或路由设备会自行判断数据流应当通过哪条路径到达对方。一旦某条动态的路径不再可用，交换设备会立即选用其他可以到达对方的路径，而这个短时的中断所造成的影响会交给通信双方运行的传输保障协议来处理，丢失的数据包会被重传。而用户对此不必关心，最多会感觉有短暂延迟，而不必重新和对方建立连接。

2. 面向无连接的优势

面向无连接的包交换网络比面向连接的网络有很多优势。面向无连接的包交换网络是网络通信的一种趋势，目前的VOIP、IPTV等应用都是想利用包交换网络来代替普通的电话交换网络和有线电视网络。

不要把“面向无连接”和“TCP是有连接的”混淆在一起。TCP是一个端到端的协议，它运行于通信双方，而不是通信所经过的网络设备上。TCP的连接不是物理连接而是逻辑连接。TCP其实就是一个状态，本身保持一个状态机用来侦测双方的数据流是否成功发送或者接收。实际通信两点间的连接可以经过包交换网络，同样也可以经过面向连接的网络。也就是说，“面向连接”和“面向无连接”是指链路层的概念，而TCP是传输层的概念。

9.1.2 串行和并行

串行传输在长距离高速传输方面，也必将取代并行传输。

并行SCSI总线就是一种面向连接的并行的共享总线技术。其趋势就是必将被高速串行的、面向无连接的网络通信技术所取代。取代之后的结果，必将使这个网络的扩展性大大增强，使存储系统和主机系统可以远隔千里进行通信。

得益于FC带来的诸多好处，现在人们终于可以摆脱存储系统和主机必须放在一起的限制了。如主机在北京，而盘阵可以在青岛，它们之间通过租用ISP的光缆线路进行连接，在这条线路上承载FC协议，从而达到主机和盘阵之间的通信。这样，在北京的主机上就可以直接认到远在青岛的盘阵上的LUN逻辑盘。

由于FC接口速度可以是1Gb/s、2Gb/s、4Gb/s甚至8Gb/s，并且盘阵前端可以同时提供多个主机接口，所以它们的带宽之和远远高于后端连接磁盘的并行SCSI总线提供的速度。这样，就可以在盘阵的后端增加更多的SCSI通道，以便接入更多的磁盘来饱和前端FC接口的速率。

9.2 不甘示弱——后端也升级换代为FC

在将主机与盘阵之间的接口、链路都替换成FC协议之后，人们不断增加磁盘阵列后端磁盘的数量，以达到前端众多FC接口的饱和速率。但是此时瓶颈出现了，后端每增加一个SCSI通道，最多能接入15块磁盘，数量太少了。增加SCSI通道也不是一个最终解决办法，能否找到一种彻底的解决办法呢？

抛弃老爷车——让SCSI搭乘高速专列

要解决这个问题，就要彻底抛弃盘阵后端的SCSI并行传输总线网络，就像当初抛弃前端SCSI总线一样。而且不仅仅要使接入硬盘的数量增加，还要高速、稳定。既然已经将前端传输网络替换成了FC交换网络，那么是否可以将后端网络也从并行SCSI替换成FC呢？理论上是完全可行的。

由于FC协议系统提供了两种网络拓扑架构，所以要考察后端存储网络到底使用哪种架构比较合适。首先交换式架构Fabric是一种包交换网络，寻址容量大，交换速度快，各个节点间可以同时进行线速交换，无阻塞通信，但是成本较高。其次是FC-AL仲裁环架构，带宽共享，每个环寻址容量128，最关键的一点是，它实现起来比交换架构简单，而且成本也低很多。

提示： 第7章讲的FC的一些登录、注册等过程，都是在Fabric架构中才发生的，而在FC-AL则是另外一套环初始化过程。

如果把每个磁盘都作为一个节点连接到Fabric交换网络中，性能绝对是无可挑剔的，由共享总线变成了点对点交换式通信，性能也会提升

很大。但是这样做，不但要在盘阵的后端实现一个FC交换矩阵，而且要在每块磁盘上实现FC拓扑中的N端口，这两部分成本是非常巨大的。为了降低成本，只能选择性能稍差，但是成本低的FC-AL仲裁环架构来连接磁盘阵列的控制器和磁盘，而且在每块磁盘上都实现FC拓扑中的L端口。

虽然这样做性能会比Fabric架构差，但是至少比并行SCSI总线强多了。对于并行SCSI总线来说，目前最高的标准是Ultra 320，裸速率320MB/s，实际最大传输率大概能有85%的效率，也就是280MB/s左右。有人做过实验，在Ultra 160（裸速率160MB/s）的总线上，按4KB数据块随机访问6块SCSI硬盘时，SCSI总线的实际访问速度为2.74MB/s，IOPS大约700次/s。这种情况下，SCSI总线的工作效率仅为总线带宽的1.7%。在完全不变的条件下，按256KB的数据块对硬盘进行顺序读写，SCSI总线的实际访问速度为141.2MB/s，IOPS大约564次/s，SCSI总线的工作效率高达总线带宽的88%。

由于FC-AL目前刚刚普及到4Gb的带宽，裸速率400MB/s，这样就比Ultra 320的带宽要高。不仅在速率上，FC在效率上也比Ultra 320并行总线要高。但是2G速率的FC-AL，其裸速率仅200MB/s，比Ultra 320低很多。而在4G的FC-AL出来之前，很多磁盘阵列就已经用2G的FC-AL替代后端的Ultra 320总线了。为何这些产品宁愿忍受2G速率FC相对Ultra 320一半的速度，也要将其后端替换为FC架构呢？

其因素主要有三个。

（1）可扩展性。受并行SCSI总线仲裁机制本身的限制，决定了一条总线上不会有太多的节点，16个节点的数量已经达到它的可管理极限了。而FC-AL仲裁环则不然，它的极限是128节点，这就比SCSI总线强

多了。这个限制的突破，使后端可以连接更多的磁盘，很容易就可以在单台磁盘阵列上实现上TB或者几十TB的容量（通过连接扩展柜）。

虽然一个FC-AL环的速度比一条Ultra 320总线低，但是多条总线和多个后端通道可以集成在一个控制器上。而并行SCSI接口和线缆都很宽大，想集成在一个小的空间上很难。FC由于是串行传输，两条线一收一发足够了。接口也很小，如SFP光纤接口，只有指头肚一样大小，可以很方便地在后端上实现多个通道。

（2）IOPS值比并行SCSI总线的架构显著增加。为什么这么说呢？我们分析一下。高IOPS通常意味着IO SIZE值比较小的情况下，如果使用并行SCSI总线，由于可接入节点数量较少，磁盘数量少，每秒可接受的IO请求就少；而FC-AL的后端，一个通道可以连接120多个磁盘，可以做成很多Raid Group，每秒可接受的IO请求就比并行SCSI多得多。所以，虽然2Gb带宽的FC网络传输在持续传输速率上比不过Ultra 320的320MB/s的速度，但是IOPS却比Ultra 320总线高很多。在特定条件下，2Gb的FCAL链路在IOPS和吞吐带宽指标上都会超越Ultra 320的SCSI总线。

现在的大部分应用都是要求高IOPS的，它们产生的一个IO SIZE一般都比较小，而且随机IO居多。但是对于视频编辑等领域，无疑是要求高传输带宽的。面对这种应用，可以通过在盘阵后端加入多个FC-AL环来解决，后端带宽总和等于环数乘以环带宽。

（3）双逻辑端口冗余。由于FC的串行方式使得数据针脚数量降低，相同的空间内很容易做成双逻辑端口冗余。双逻辑端口磁盘可以有效保证当其中一个端口发生故障之后，磁盘可以继续使用另外一个备用端口接受IO请求。

9.3 FC革命——完整的盘阵解决方案

FC在盘阵的前端接口技术的革命成功之后，又在后端的接口技术上取得了成功。FC技术的两种拓扑，一个称霸前端，一个称霸后端，在磁盘阵列领域发挥得淋漓尽致。

与此同时，磁盘生产厂家也在第一时间将FC协议中的L端口和FC硬件芯片做到了磁盘驱动器上，取代了传统的SCSI端口。同时，根据FC协议的规定编写了新的Firmware，用于从FC数据帧中提取SCSI指令和数据，完成FC协议通信逻辑。

提示： 并行SCSI磁盘以及其他设备目前仍有比较广泛的应用，尤其是服务器本地磁盘。服务器本地磁盘一般只安装操作系统，一般情况下应用数据都会放到SAN的磁盘阵列上，所以对本地磁盘的性能要求不高，使用Ultra 320磁盘足矣。另外，普通独立磁带机一般也用Ultra SCSI 320作为其外部接口。只有在大型磁带库设备上，为了将其接入FC SAN才会使用FC接口。

因为要把每块磁盘都连接到FC-AL网络中，所以磁盘上要做上一个FC接口。由于磁盘阵列背板需要连接众多的磁盘，所以就注定不可能用柔软纤细的光纤来连接磁盘到背板，必须使用硬质铜线，让磁盘的FC接口用铜线来接触盘阵背板上的电路，这样才能做到方便地插拔。

9.3.1 FC磁盘接口结构

FC磁盘的接口为SCA2形式的40针插口，如图9-1所示。

图9-1 SCA2接口规格

从图9-1中可以看出，FC磁盘的接口与Ultra 320 SCSI磁盘的接口形状完全相同。但是在接口上的物理信号定义和承载的上层协议是完全不同的：FC磁盘接口是承载的FC-AL协议，而SCSI磁盘接口承载的是并行SCSI总线协议。图9-2是FC SCA2型接口的信号定义表。

图9-2 FC接口SCA2针脚信号定义表

9.3.2 一个磁盘同时连入两个控制器的Loop中

图9-3和图9-4分别给出了一串磁盘以单Loop和双Loop接入的情形。

图9-3 单Loop连接示意图 图9-4 双Loop连接示意图

从图中可以看出来，原来这个物理接口中共包含了两套逻辑接口，可以分别接入一个FC-AL的环路中。

9.3.3 共享环路还是交换——SBOD芯片级详解

革命之后，被并行SCSI总线技术禁锢多年的存储系统终于解放了，迎来了全FC架构的春天。

各个厂家分别推出了自己的产品。由于虽然一条FC-AL环最多能接120多块磁盘，但是有人测试过，环上节点的数量在最大值的二分之一时，性能达到最大化。再增加节点数量，性能不升反降。究其原因，可能是因为60个左右的节点，已经达到FC-AL环的仲裁性能以及带宽共享限制的极限了，如图9-5所示。如果再增加节点，那么用于仲裁所耗费的资源，就会影响性能的发挥了。这也是仲裁环或者总线的通病。对于

Fabric架构，就没有这种限制。

图9-5 FC-AL环的性能曲线

思考：节点数量和仲裁/帧转发效率是一对矛盾系统，只能在二者之间进行折中选择。在60节点数量左右，能达到最大IOPS。如果还想增加节点数量，增加盘阵所提供的总容量，那就只能牺牲IOPS性能。同样，每个环上的磁盘数量也不能太少，太少的话将达不到最高IOPS，虽然此时仲裁和帧转发速度快。

有没有可能将后端的共享FC-AL环路架构，改变为交换式架构呢？改为交换式架构是可以，但是不能改为Fabric，因为其成本相对偏高，而且Fabric的一些特性对于后端来说是用不到的。

Emulex公司发布的InSpeed SOC422芯片、PMC-Sierra公司的PM8378芯片等就是运行FC-AL协议但物理架构是交换架构的芯片。而这个交换架构绝非Fabric，因为其遵循的上层逻辑依然是FC-AL逻辑，只是在物理连接上用点对点交换架构，替代了“节点大串联”的Loop结构，使节点与节点之间传输的数据可以通过交换矩阵直达，而不是在环路上一跳一跳的中继。然而，这些芯片依然可以用在Fabric交换机上，只要经过一定改造并且在上层运行对应的Firmware即可。

其实就是用星型连接取代串联，而电路运行的逻辑依然是FC-AL仲裁过程，因为位于控制器上的FC适配器依然会执行FC-AL仲裁等逻辑，只不过这个仲裁过程变得非常简单，不再需要所有磁盘参与，而由这块芯片来进行仲裁。此外，某节点同一时刻依然只能与一个节点通信，节点感觉不到底层电路架构的变化。由于同一时刻还是只能存在一对节点进行通信，所以链路带宽依然是共享的。因此，这种交换架构做的并不

彻底，它没有过渡到包交换或者所有端口无阻塞全交换架构，虽然物理上已经可以实现点对点的通路。

图9-6所示的为传统FC-AL架构和半交换式FC-AL架构示意图。我们可以看到左边的拓扑完全就是磁盘串，而右边则变成了星型的架构，中间由一个ESH交换模块辐射出来。

图9-6 传统FC-AL环和半交换式FC-AL环

目前高端盘阵后端控制芯片几乎全部采用半交换式架构了。半交换式架构有以下好处。

- 控制节点与所有其他节点间都是点对点矩阵直连架构，相对于环路架构减小了数据传输时的延迟。数据帧不需要一跳一跳的转发，可以直接从发起者到达目标。
- 可以快速侦测和隔离某个节点的故障而不影响其他节点。
- 由于降低了链路延迟，增加了效率，所以在性能可接受的前提下，一条链路可接入的节点数大大增多。
- 相对于纯环路架构，半交换架构提高了传输速度和IOPS。

纯环路架构的扩展柜中，连接硬盘的背板只是一个FC-AL环路连接装置，而升级到半交换架构的硬盘扩展柜的，其连接硬盘的背板上就有了Switch芯片（其实这个芯片一般存在于从背板单独接出的扩展模块上），可以级联多个扩展柜。在逻辑上，这些级联扩展柜中的所有磁盘属于一个逻辑上的Loop，前者被称为JBOD（Just Bunch Of Disks），后者被称为SBOD（Switched Bunch Of Disks）。

提示： 关于这些模块的实物图，请参照本书第6章的相关章节。

1. PMC-Sierra公司PM系列芯片简介

下面以PMC-Sierra公司的PM8368和PM8378两款芯片为例来详细解释一下这种芯片的作用方式。

1) PBC芯片

在传统模式的FC-AL架构下，所有环路上的节点（adapter和磁盘）都通过串联架构串接到了一起。针对这个架构的PM8368芯片是一款具有18个2Gb/s的FC接口的带有PBC（Port Bypass Control）功能的芯片，相当于一个Loop串联器，并且带有端口Bypass功能，可防止某个端口故障引发的全Loop断开。图9-7所示为PM8368芯片的方框图。其18个端口中，通常用16个连接磁盘，其他两个连接上位芯片和（或）下位芯片。这种不带有交换逻辑的普通Loop芯片，业界一般称为PBC芯片，即旁路控制芯片。

图9-7 PM8368芯片方框图

图9-8所示的是将PM8368芯片与PM8372芯片（一款只有4个FC口的PBC环路集线芯片）搭配在一起而实现的扩展柜级联，每扩展柜有16块盘。引入PM8372芯片的原因是将其作为一个二级级联桥来减缓一级芯片（PM8368）的端口耗尽问题。一级芯片需要连接最终硬盘等设备。

图9-8 利用PM8368芯片与PM8372芯片级联扩展柜

2) CTS芯片

图9-9所示为PM8378芯片的方框图。此芯片为新一代的CTS芯片，CTS为Cut Through Switch的简称。这款芯片具有交换逻辑，IO性能相对

于PBC芯片大大增加，而延迟大大降低。这块芯片可以接入18个4Gb/s的FC接口设备。在上部的模块中，可以看到三个模块：Arb Mgmt（仲裁管理模块）、AL_PA Table（AL-PA地址端口映射表模块）、Cut Thru Mgmt（捷径交换模块）。

图9-9 PM8378芯片方框图

- Arb Mgmt（仲裁管理模块）：前面曾经说过，一个交换芯片必须感知FCAL的仲裁逻辑，才能作为一个中心仲裁器总揽仲裁权，而不用Loop上所有的节点都参与进来。这个仲裁管理模块实现的就是这个功能。FC适配器只要发出仲裁请求，芯片就会马上通过，适配器立即就获得了使用权，加快了仲裁的速度。如果某时刻多个设备都发起仲裁，则芯片会根据自己的逻辑来处理这些请求，比如根据优先级等。
- AL_PA Table（AL-PA地址端口映射表模块）：传统的FCAL环路上，数据从通信源被发送到对应AL-PA地址的目的设备，数据帧需要一站一站地向下接力传送。交换芯片的另一个作用就是抛弃这种低效的传输方式，使数据可以一站直达目的。芯片使用一张映射表来维护交换逻辑，这就和以太网交换机维护一张MAC-端口表一样。FCAL交换芯片每收到一帧数据，就会根据这个AL_PA Table来查找帧中的AL-PA地址所对应的芯片针脚是哪一个，然后直接将此帧转发到对应的针脚上，也就传送到了针脚所连接的磁盘FC接口上。
- Cut Thru Mgmt（捷径交换模块）：这个模块其实就是执行交换过程的。每当有帧需要交换，这个模块就会发送一个信号到图9-9中部的那个矩阵上，改变矩阵当前的通路布局，从而将数据从源传向目的设备。

图9-10是利用PM8378芯片实现的磁盘扩展柜级联示意图。18个端口中有16个连接了磁盘，另外两个分别连接了上位和下位芯片。

图9-10 扩展柜级联示意图

还可以将PBC芯片和CTS芯片组合在一起级联扩展柜。如图9-11所示，为了降低成本，使用了18口的PBC芯片来连接磁盘，然后用一个4端口的CTS芯片来作为二级级联桥。在这种情况下，每个扩展柜中的磁盘之间为普通Loop串联，但是扩展柜与扩展柜之间却是交换拓扑，这样就相当于把整个Loop上的节点划分成域，每个扩展柜就是一个共享域（冲突域）。现在流行的做法是：磁盘连接到CTS芯片中，而柜子与柜子之间是PBC芯片，所以图9-11所示的恰好相反，这一点请读者注意。

图9-11 PBC与CTS的组合

如图9-12所示，这种方式使用了高成本的CTS芯片组成了全交换架构的Loop。整个Loop上的节点之间都形成了交换架构，但这个交换架构并不是无阻塞多路同时交换的架构，依然是同时只允许两点间通信的上层FC-AL逻辑的架构。

图9-12 全CTS架构

SBOD控制模块上的芯片不只有PBC或CTS，还包括其他各种功能的芯片，在图中并没有给出，下面会一一介绍。

2. SCSI Enclosure Service简介

SCSI Enclosure Service简称为SES，从字面上理解就是SCSI协议中用于查询磁盘扩展柜（Enclosure）各种状态的一种服务（协议）。扩展

柜上有很多组件需要被监控，比如电源模块、风扇散热模块、各种指示灯、温度传感器等。这些组件都通过某种总线（比如I2C、GPIO）方式连接到某个芯片，然后这块芯片再通过I2C等连接到单片机或CPU。或者通过外部传感器直接连接到高集成度的单一芯片上。SCSI客户端程序（也就是SCSI Initiator端程序）通过发送SES协议规定的各种指令来查询Enclosure上各个组件的状态信息。

SES服务模块就是指扩展柜上的CPU，其中运行着处理SES的代码。这个CPU可以是主控CPU，也可以是独立的芯片。存储设备机头主控制器有两种方式将SES指令传输给这个CPU，如下所述。

1) 独立SES服务模块

如图9-13所示，独立的SES服务模块独占一个LUN ID（十六进制的0D），可以直接寻址。也就是说，SES服务模块就相当于一个FCAL Loop上的ID，机头直接将SES指令传输给这个ID。对应电路层面，主交换芯片收到目标为此ID的帧，便会直接转发给这个ID所连接的设备，这里就是SES处理芯片。

图9-13 独立SES服务模块示意图

2) 附属的SES服务模块

如图9-14所示，附属的SES服务模块往往会利用一个已经存在的设备的地址来与自己共用。而物理接口上，这个设备一般会有一条旁路来连接到SES服务模块，比如目前FC磁盘普遍使用的SCA2接口中就有对应的ESI（Enclosure Service Interface）接口来连接到SES服务模块上（CPU）。存储设备的主控机头首先将一条SES可用性探询指令发送到这个已经被磁盘占用的ID，如果这个ID对应的槽位上有连接设备，且该

设备支持转发SES帧，则该设备会返回一个确认帧，帧中携带有特定Page的值，主控机头收到之后便会分析出这个设备是否支持SES转发。机头随后发出纯SES指令，磁盘收到之后，会通过ESI接口将SES帧发送给ESI接口的对端，当然就是SES处理芯片了，芯片收到SES指令之后，便通过I2C（或者其他类型总线）总线去查询外部传感器的信息，然后封装成SES返回帧，通过ESI接口发送至磁盘，磁盘再转发给主控机头。磁盘的Firmware必须支持转发SES帧。

图9-14 附属的SES服务模块示意图

3. SBOD上的CPU

1) PMC-Sierra公司的PM8393芯片

这是一款基于MIPS32架构的单片机，内建有128KB的RAM，外部时钟为106.25 MHz，可以将其连接到上文所属的PM8378 Switch芯片中。这款芯片又名Sotrage Management Controller，简称SMC。图9-15为其方框图。

图9-15 PM8393芯片方框图

图9-16是PM8393芯片充当扩展柜总控CPU，用PM8378/PM8379充当交换芯片来连接磁盘，再加上其他一些芯片共同组成的一个完整的扩展柜控制模块架构。

图9-16 用PM8393做为扩展柜总控CPU

2) Qlogic公司的GEM359芯片

图9-17～图9-19所示为这款芯片的方框图。

图9-17 GEM359芯片方框图（1）

图9-18 GEM359芯片方框图（2）

图9-19 GEM359芯片方框图（3）

4. SBOD上的ROM和RAM

SBOD扩展柜上的CPU需要执行Firmware中的代码来完成一系列的动作，如SES服务等。所以，Firmware编写的质量直接决定了扩展柜是否能向机头上的控制器稳定和迅速地报告扩展柜上所发生的一切事件。

Firmware一般存在于扩展柜控制模块上的ROM芯片（如Flash芯片）中，并且可以随时将升级之后的Firmware通过FC接口直接写入芯片。这种直接通过实际数据链路来升级系统控制数据的方式叫做in-band升级，即“带内升级”。如果用一种单独的通道来访问Flash芯片并做升级动作就叫做out-band升级，也就是带外升级。

SBOD控制模块上一般都有外置的RAM芯片，有些内置了RAM的单片机除外。

5. PATA、SATA和SAS磁盘怎么办

PATA（IDE）盘和SATA盘相对于FC盘和SAS盘来说，成本降低了很多，且可以实现高容量，现在已经有了1TB的SATA磁盘。对于一些对IO性能要求不高的环境来说，使用SATA盘无疑是很合适的。但是面对不同的接口，不同的指令，单独对这些磁盘实现一套盘柜和控制器体系实在是不方便。且现在企业都要求高度整合，统一分配，方便管理。

根据这个需求，各种适配器和转换逻辑出现了。图9-20所示为一个SATA-SCA2接口转换器。SATA磁盘只要接上这块PCB，就可以从物理上融入FC盘柜中，也就变成了所谓的FATA盘。除此之外，还有很多其他类型的转接电路，比如PATA-SCA2、SAS-SCA2等。

图9-20 SATA-SCA2转接电路板

物理上融入了，在逻辑上也需要进行转换。SATA磁盘使用ATA指令系统，FC和SAS磁盘使用SCSI指令系统，二者不兼容。所以需要有一个中央芯片负责在两种逻辑之间互相转换。

Sierra Logic的SR1216芯片是一款高集成度的芯片。这款芯片将ATA-SCSI转换逻辑以及两个MicroProcessor做到了一块单一的芯片中。MicroProcessor就是微型CPU，用来运行外部Flash芯片中的Firmware，从而实现SES等扩展柜管理程序。

图9-21所示为SR1216芯片实物图。

图9-21 SR1216芯片实物图

图9-22和图9-23所示为用SR1216来充当SATA桥和总控CPU所形成的扩展柜控制模块架构图，PBC表示FC Loop旁路控制芯片，SES Processor表示用来收集外部传感器的专用芯片。

图9-22 SR1216架构的扩展柜控制模块
(1)

图9-23 SR1216架构的扩展柜控制模块
(2)

另外，PMC-Sierra公司也有多款多端口SATA复用芯片，不过有一些需要搭配额外的ATA-SCSI转换桥芯片。图9-24为其示意图。

图9-24 PMC-Sierra的SATA盘柜控制模块架构图

9.4 SAS大革命

SAS技术是近一两年来才被普遍使用的技术，其相对FC技术有多种优点，也有其缺点。本节向大家介绍一下SAS技术的底层架构。

9.4.1 SAS物理层

适用于存储系统的网络不止FC一个，同档次的网络传输系统还有一个叫做SAS的，全称为Serial Attached SCSI，即串行SCSI。FCP也属于串行SCSI，SAS只是一个名称，不要太较真。SAS是于2001年被Compaq、IBM、LSI Logic、Maxtor和Seagate联合提上日程的。大家都知道，现在普遍用于PC的SATA硬盘，也是从2001年之后才逐渐崭露头角的。的确，当时几大厂家在开发串行ATA时就考虑到：为何不将SCSI一同纳入开发范围呢？于是SAS便悄悄的被开发了。正是由于这种因果关系，今天普遍用于PC服务器和小型机的本地硬盘的SAS磁盘的接口形状与SATA盘是相同的，只是比SATA盘多了一个数据接口，像FC磁盘一样用来连接两路控制器。第二数据接口位于第一数据接口靠背的位置，需要翻过来才能看到，如图9-25所示。

图9-25 SAS和SATA接口的区别

SAS网络与FC有一个本质区别，即SAS为全交换式架构，不像FC一样有Fabric和FCAL两种架构。如果让一个控制器（Initiator）和多块磁盘作为网络节点直接连接到SAS网络中的话，那么控制器和所有磁盘之间都是全双工线速无阻塞交换的，控制器可以直接向任何一块磁盘收发数据；同样，磁盘也可以在任何时刻直接向控制器发送数据。这一切过程都是没有冲突的，也不需要像FCAL那样的仲裁机制，而且每个节点

都独享传输带宽，系统整体带宽=节点数量×每个节点享有的带宽。

在SAS网络中，每一个物理接口都需要有一套底层编解码器负责对数据进行8/10b编解码。每个物理接口又称为PHY，即Physical的简写。SAS 1.0时代，每个物理接口的编解码速度为3Gb/s，SAS 2.0时代（2010年初才有产品正式宣布），这个速率被提升至6Gb/s。换算一下就可得知，SAS 1.0的每个接口数据带宽为300MB/s，SAS 2.0则翻倍。相比之下，同时代的FC接口物理速率已经普遍为4Gb/s，而且8Gb/s的产品已经上市，SAS慢了一小步，而且还相差2Gb/s。但是SAS的一项设计却比FC走到了前面，即可以将多个PHY捆绑成一个逻辑接口，数据并行地在多个PHY中传输，就像PCI-E一样，每个PHY速率2.5Gb/s，4XPCIE便是4个PHY捆绑。4X的SAS接口，其速率就变成了12Gb/s（SAS 1.0）或者24Gb/s（SAS 2.0）。而目前FC是做不到这一点的。多个PHY经过捆绑之后形成的逻辑端口称为“宽端口”，不捆绑的独立单一PHY称为“窄端口”。宽端口一般用于主机SAS适配器连接SAS接口外置磁盘阵列时使用，因为宽端口可以提供更高的带宽，消除瓶颈，同时，连接使用的线缆也借用了Infiniband网络的设计。如图9-26所示为一个4X宽端口所用的线缆接头实物图。

图9-26 SAS的4X宽端口

图9-27所示的是一块主机SAS适配卡的连接示意图。此适配卡有两个SAS通道，每个通道又有4个PHY，其中一个通道用于连接主机箱内部的SAS或者SATA磁盘，通过线缆连接到转接背板上，然后4块磁盘可以以热插拔方式插到背板上对应的接口。另一个通道使用上文所述的4X SAS线缆连接外置SAS接口的磁盘阵列。如果在这块SAS卡上设计有独立的Raid芯片或者直接集成到SAS IO Processor中，那么这块卡就变为了一块可接SAS或者SATA硬盘的Raid卡了。

图9-27 SAS适配器内部连线示意图

由于SAS和SATA硬盘的接口是相同的，所以自然想到是否SATA硬盘也可以接入SAS网络呢？答案是肯定的。SAS协议利用STP（SATA Tunneling Protocol）来兼容SATA协议，对SATA节点的数据收发，SAS是将数据封装在SAS协议帧中传递的，数据到达SATA节点后，解封封装，然后再由SATA节点处理。这一点FC自身是无法做到的，而需要一个SCSI-ATA协议转换器（前文所述的SR1216芯片）以及一个SATA-FC接口转换器（前文所述的SATA-SCA2转换板）来实现。

提示：对于物理接口转换器，有一点需要了解，即SATA盘只有一个数据接口，而FC磁盘有两个逻辑数据接口（包含在一个物理SCA2接口中），所以接口转换器又被称为Port Multiplexer，即前端实现两个逻辑接口（对于FC来讲）或者两个物理PHY（对于SAS来讲），而后端连接同一个SATA物理接口，转换器将前端的所有接口都映射到后端的一个接口，并且在前端虚拟出两个物理接口地址以用于前端网络的寻址操作。这种接口转换板又被某些厂家称为“Dongle”。

SAS作为一个交换网络，那么理所当然的就应该有对应的交换媒介，即交换机，或者交换芯片。让我们来看一张PMC-Sierra公司所设计的一款36端口SAS交换芯片（PM8387）的架构，如图9-28所示。

图9-28 SAS交换芯片架构

图中XBAR代表Crossbar，即这款芯片内包含一个基于Crossbar架构的交换矩阵用来在各个PHY之间进行高速交换。XBAR左侧连接了36个PHY，每个PHY前端是8/10b Serdes（串化解串化器），由于数据在芯片

内部都是并行传递的，到芯片外部之前都需要经过串化过程和8/10b编码过程，而从芯片外部到内部的数据则执行相反过程。SXL表示SAS Expander Link，其中Expander就表示SAS交换芯片，因为交换芯片可以让众多节点所连接并且通信，所以又叫做Expander，即扩展器。芯片中包含的另一个部件为一个32b的MIPS核心处理器，这个处理器执行Firmware以实现SES、SMP（下文描述）和其他一些外部信号（比如指示灯、Debug接口、监控）处理工作。XBAR右侧为各种信号总线和接口。XBAR中的Arbiter，即仲裁器，负责协调各个PHY之间发起的通信并控制Crossbar将数据交换到正确的目的。这里不要被其名称所误导，SAS不使用仲裁共享方式，具体数据收发方式后文描述。

9.4.2 SAS链路层

对于SAS的链路层协议内容，比如帧的组成结构、帧同步、链路错误恢复等由于太过底层，请想了解的读者自行参考SAS协议文本（大部分帧结构都与FC类似）。这里只简要介绍一下SAS链路层与FC的一个最大的不同之处以及速率适配方面。

SAS在链路层使用面向连接的交换技术。Initiator向Target发起通信之前必须建立好连接，但是要与传输层的端到端连接相区别开（SAS的传输层也有连接，下文描述）。SAS链路层的连接指的是在一个SAS网络内的两个PHY的通信路径中的所有Expander内部，都将为这条通信保持相应的资源，比如Crossbar矩阵内的交换路径。某个Initiator和Target对初次收发数据时，Crossbar上的Arbiter，或者叫Connection Manager会检测并且在Crossbar内新建一条路径，以便将它们之间的数据通过恒定的路径发送到下一条Expander的端口，下一条Expander再做相同的动作直到数据达到最终Target。连接是在每个PHY之间建立的，如果使用宽

端口，则每个PHY之间都会为对应的OPEN请求建立连接，因为每个PHY之间都有链路，所以需要保留对应的资源和路径。

具体连接发起和结束的过程如下。

(1) Initiator端的链路层首先通过PHY接口向其连接的Expander发起一个OPEN帧，帧中携带的是Target端PHY的SAS地址。

(2) 本地Expander收到之后，首先返回AIP帧（Arbitration In Progress），这期间，Expander会为这个请求分配资源。由于Expander需要维持很多Initiator和Target的连接，有时资源不够的话，分配的资源会相冲突，需要根据SAS地址来判断连接的优先级以便有限分配高优先级连接的资源，这个过程就是仲裁，但是它与FCAL的仲裁有着本质的不同。

(3) Arbitration成功之后，在Expander内为其保留恒定资源和Crossbar路径，然后将这个帧路由到Target所在的Expander，途中经过的所有Expander都会为这个初始连接分配恒定资源和路径。

(4) Target端的链路层接收到OPEN请求之后，向Initiator端返回一个OPEN_ACCEPT帧。连接建立。

需要关闭连接的时候，Initiator端发送CLOSE帧。

图9-29所示为连接发起和接收期间的步骤。

图9-29 SAS连接发起和接收过程

对于适配不同速率的设备，SAS的链路层通过在快速链路上插入对应长度的ALIGN冗余数据来保持速率适配，如图9-30所示。

图9-30 SAS的速率适配

9.4.3 SAS网络层

作为一个网络，SAS当然也需要编址和寻址。SAS网络中的每个节点使用的地址与FC网络类似，也是一个64b长度的定长地址，也叫做WWN；但是与FC Fabric不同的是，SAS直接使用这个64b地址来路由数据包，而FC Fabric则使用另外分配的24b的Fabric ID来路由数据包。编址是为了寻址，寻址就需要有地址表，或者叫做路由表。每个Expander上运行着一种协议，用来执行整个系统中的Expander发现以及路由条目学习，这种协议叫做SMP，全称为Serial Management Protocol。

我们首先来看只有一个SAS交换机或者交换芯片（或者SAS Expander）的情况下，SAS Expander是如何寻址的。熟悉IP路由的人很容易地就可以理解，对于直接连接在Expander上的终端节点，就属于直连模式，Expander在获取到它们的地址之后，会将这些地址加入路由表，并标明路由条目的模式属于“D”类型，即Direct。如图9-31所示，一个12端口的SAS Expander上接入了4块SAS磁盘，SAS磁盘为终端设备，Expander会将这4条记录收录到路由表中，并标明为D类型。图中的SMP表示Serial Management Protocol，SMP在这里是一个抽象的对象，它作为一个实体路由协议和管理程序运行在每个Expander上，所以这里用它来代表每个Expander。由于目前的Expander是系统中唯一的一个，所以编号为SMP0，并且对应了一个虚拟的PHY12（第13个PHY并不存在）。

图9-31 SAS的寻址

此时，我们再增加一个Expander，将这两个Expander级联起来，并

且在第二个Expander上也连接4块SAS磁盘。此时的路由表如图9-32所示。

图9-32 SAS级联时的路由表

上图中出现了一种新的路由类型，T类型，意思是Table Routes，暗指这种类型的路由条目是通过路由协议学习而得来的，并非自己本地直连的路由条目。SMP1表示新加入的Expander本身。然而，是否觉得缺了点什么？是的，磁盘和磁盘之间是不会通信的，磁盘只是作为Target端来等待Initiator端来发起指令。很显然Expander不是这个Initiator。我们需要向这个由两交换机（芯片）组成的网络中添加一个SAS Initiator，如图9-33所示。

图9-33 SAS Initiator的引入

上图中，Expander0的路由表中增加了一个条目，即它所连接的有一个直连终端，HOST0上的SAS适配卡，路由类型为“D，S”，D为直连类型。S即Subtractive，意义为默认路由，即，如果某个数据帧无法从D或者T类型的路由条目中找到目的地址，那么统统向S类型路由条目对应的PHY接口转发出去。SAS卡和Expander0之间同样运行着SMP协议，Expander0会学习到SAS卡为Initiator端，所以，Expander1也会学习到这条路由，但是Expander1的路由表中的默认路由显然应该设置为SMP0，即将具体路由未知的数据帧从连接SMP0的PHY转发出去。另外，Expander1也会同时学习到HOST0路由条目，但是不将其作为S类型，因为S类型路由存在的意义是可以在路由表中不保存HOST端的条目，在主机数量很大的时候，有利于保证路由效率。

SAS协议规定，可以由多个Expander任意连接而组成一个SAS网

络，但是这个网络可以接入的最大的终端节点不能超过128个，这个一级SAS网络中所有的Expander组成一个Expander set。如果要达到更大的接入容量，需要一个核心Expander，这个Expander可以最多具有128个PHY，可以接入最多128个上述的Expander Set。此核心Expander称为“Fanout Expander”，Expander Set中的Expander称为“Edge Expander”。这样，最大的SAS网络中可接入的总容量为 128×128 再减掉Expander互联所耗费的端口数。图9-34所示为Fanout Expander与Edge Expander的互联示意图。Fanout Expander作为整个SAS网络的核心，其上不允许再有S类型的路由，只能有D和T类型的路由。Edge Expander连接Fanout Expander的PHY，对应这个PHY的路由皆为S类型路由，因为Edge Expander只能将未知条目转发给核心。

图9-34 Fanout与Edge的互联

每个独立的SAS网络被称为一个Domain。

9.4.4 SAS传输层和应用层

SAS网络目前有3种应用协议：一是SSP（Serial SCSI Protocol），二是STP（SATA Tunneling Protocol），三是SMP（Serial Management Protocol）。说它们为应用层协议既恰当又不恰当，因为相对于SAS本身来讲，它们确实是处于应用层，但是相对于整体系统来讲，它们又都处于传输层（SSP和STP）或网络层（SMP）。这三种协议在传输数据之前都需要首先由Initiator向Target端发起连接请求，这种上层的连接请求也同时被映射到了链路层，链路层也会建立相应的连接，如前文所述。

1. SSP

SSP是一套用于在Initiator和Target之间传输SCSI指令的传输保障协议，与FC中的FCP层充当相同的角色。我们知道SCSI3规范已经将SCSI上层指令与底层传输系统相分离，SSP在此就作为SCSI指令的传输系统，SSP会保障SCSI指令和数据以及对指令的响应成功地被传送和接收。图9-35所示为SSP的帧结构。图9-36所示为SSP的帧类型，包含COMMAND、TASK、XFER_RDY、DATA、RESPONSE五种类型。

图9-35 SSP的帧结构 图9-36 SSP的帧类型

图9-37所示为SSP执行TSAK管理任务时所发送的指令和返回流程步骤。图9-38所示为SSP发起SCSI指令时的步骤。

图9-37 SSP设备控制指令步骤 图9-38 SSP发起SCSI指令时的步骤

图9-39所示为SSP执行SCSI Read指令时的步骤。图9-40所示为SSP执行SCSI Write指令时的步骤。

图9-39 SSP执行SCSI Read指令时的步骤 图9-40 SSP执行SCSI Write指令时的步骤

2. STP

STP是一套用于在Initiator和Target之间传送SATA指令的传输保障协议。由于SATA协议与SCSI协议完全是两套上层协议，不仅指令描述方式和结构不同，而且在底层传输的控制上也不同，所以STP就是将SATA协议的底层传输逻辑拿了过来，并将其承载于SAS底层（物理层+链路层+网络层）进行传输。其实这里说SATA Tunneling也具有一定的误导性，可能会误认为SCSI就不需要Tunneling。其实不管是SCSI还

是SATA，它们都是以不同程度的Tunnel+Map的方式被承载于SAS网络上传递。可以参考本书后面章节来了解协议之间的相互作用问题。图9-41所示为STP执行SATA指令时的步骤。

图9-41 STP执行SATA指令时的步骤

3. SMP

SMP是一种对SAS网络内所有Expander进行管理的协议，包括Expander拓扑发现和路由协议。由Initiator端建立连接后向Target端发起查询请求，Target只能响应Initiator的查询而不能主动向Initiator发送消息。这里大家可能有个疑问，即不管是SSP中的Target还是STP中的Target，它们都是有实在的物理存在的东西来对应的，比如SSP的Target就是某PHY后面的SAS磁盘，STP的Target就是某PHY后面的SATA磁盘。那么SMP中的Target到底是什么呢？上文讲SAS路由表时曾经提到过一个Expander上的虚拟PHY，对应了一个SMP设备，其实SMP的Target就是整个Expander本身，Expander接收到针对这个Target的消息，就会自己做处理而不是转发到其他某个PHY。图9-42所示为SMP的请求和查询步骤。

图9-42 SMP的请求和查询步骤

9.4.5 SAS的应用设计和实际应用示例

SAS Expander芯片有多种Phy接口数量可供选择，比如16/24/32/36等。提供不同的Phy数量是为了满足磁盘阵列厂商所设计的不同盘位的扩展柜。每个SAS磁盘扩展柜里面基本上有两个SAS Expander，这两个Expander各自连接到这个柜子中的所有SAS磁盘，每个SAS盘两个SAS

Phy。这两个Expander还分别要连接到上行和下行的其他扩展柜或者机头，最顶上的是机头，机头里也要有Expander（做Table Routing），用来连接SAS控制器芯片以及扩展柜里的Expander。机头中的Expander相当于一个总桥，总桥再分别出4个Phy连接到上行外部主机和下行扩展柜中的Expander，12盘位的可以估算它的扩展柜单片SAS Expander芯片接口规格： $12 + 4 \times 2 = 20$ 口芯片，24盘位的则使用36Phy的SAS Expander芯片。扩展柜与扩展柜之间用SAS线缆连起来，一般都是4个Phy并联的宽端口。

如图9-43所示，在磁盘扩展柜中使用SAS Expander而不是FC Loop Switch芯片，其他设计保持相同，那么一款SAS扩展柜就被打造出来了。多个扩展柜通过Expander间的级联就可以扩大到比较可观的容量。主机接口以及Expander间接口都使用4X宽端口以保持级联带宽，这样，系统吞吐量上也可以保持高水准。总体来讲，SAS比FC实惠、量足。

图9-43 SAS控制器

如图9-44所示为HDS公司AMS2000系列磁盘阵列的后端架构图，很明显它使用了PMC公司的Expander芯片。

图9-44 AMS2000的控制器后端

9.4.6 SAS目前的优势和面临的挑战

1. SAS目前相对于FC的优势

SAS能够使用宽端口来提供较高的带宽，这一点已经可以与8Gb FC接口媲美并且超出了后者50%的速率。另外，SAS使用STP协议来承载SATA协议，使得SAS和SATA磁盘可以同时存在于同一个存储系统中，

甚至可以混插在同一个磁盘柜中，这一切都不需要上层做过多的更改，只要在Initiator端的上层协议处理模块中加入ATA支持即可，或者统一使用SCSI，然后在其下层增加一个SCSI-ATA翻译层。FC在理论上当然也可以被开发承载SATA协议，但是目前FC尚未提供类似FCP的映射层来适配SATA协议。

SAS相对于FC来说，其成本较低，体现在SAS Expander和SAS磁盘上。目前一片12端口3Gb/s的Expander芯片的价格大概在35美元左右。而SAS硬盘在盘片、转速、机械臂、磁头等的设计参数和用料上与高端FC硬盘相同，唯一的区别就在于外部接口使用的协议不同，所以设计也不相同，但是SAS磁盘却比FC磁盘廉价。由于磁盘主要的瓶颈在于平均寻道时间和平均旋转延迟以及转速，加上存储控制器对后端所有磁盘的Raid化和虚拟化之后，实际使用的时候，在外部接口速率相差不大的情况下，磁盘接口速率对系统整体性能影响不大。

2. SAS目前面临的挑战

SAS的一个最大挑战是其连接线缆。目前的产品都是使用铜缆，这就导致其传输距离非常受限，通常最大距离被限制在10米。而且线缆较粗较硬，不利于布线。这种限制直接限制了SAS的发展。比如实现前端网络化的时候，多台主机与多台磁盘阵列都需要连接到SAS交换机（Expander）上，而由于较粗较短较硬的线缆的限制，SAS交换机就显得很尴尬，所以SAS交换机也一直没有实际产品。这些因素导致了SAS目前被应用的范围非常窄，比如，普遍应用于主机本地内置磁盘。在用于磁盘阵列设计的时候，主要使用在后端，而前端依然为FC，比如HDS公司AMS2000产品。但是也有一些前端后端都使用SAS的产品，但是这些产品均被定位于低端，比如IBM DS3200，由于前端只能连接一

台或者两台主机（一个或两个SAS宽端口），其扩展性被限制，所以只能被定位到低端。

SAS的另外一个挑战是其单PHY的速率，当前普遍使用的SAS接口磁盘速率为单口3Gb/s，相对于FC的4Gb/s来讲还是相差了25%。但是，目前6Gb/s的单PHY速率刚刚露头并有了实际产品，令人欣慰的是，在尚未有8Gb/s FC接口的磁盘产品出现之前，已经出现了SAS 6Gb/s接口的磁盘，这充分说明SAS的发展是飞快的，而FC又处于了一种是走是留还需要观望的状态之中。而SAS下一个目标是单PHY速率12Gb/s。照SAS目前的发展速度，FC的卫冕之路将会更加艰难。

不过，在2010年新发布的存储系统比如HDS VSP及IBM Storwize V7000，以及包括EMC的新一代CX存储系统中，FC磁盘已经悄悄地从这些存储系统的支持列表中消失了。但是存储前端供主机访问的网络类型，FC依然处于主导地位。相信在光连接的SAS出现之后，前端也有可能彻底被SAS革掉。FC革掉了SCSI，SAS又把FC从后端革掉了。

9.5 中高端磁盘阵列整体架构简析

图9-45所示为NetApp的FAS6000系列实物图，其中间部分为两个机头，其余均为扩展柜。

图9-45 NetApp公司FAS6070磁盘阵列

中高端盘阵一般都会配有两个控制器，不但可以作为冗余，而且可以分担后端不同的环路。图9-46是一个典型的双控制器，且前后端均为FC架构的磁盘阵列拓扑示意图。

图9-46所示的机头（控制器所在的机柜）中不包含磁盘。实际产品中，有些由于控制器主板比较小，机头本身也可以放入若干磁盘。但是有些高端产品的控制器主板做得比较大，IO适配器比较多，再加上电源模块和风扇模块，造成机头内部空间不足以放下多余的磁盘。

图9-46 全FC架构磁盘阵列

如图9-46所示，每个控制器上有4个前端FC接口和4个后端FC接口。每个后端FC接口可以连接到一个FC-AL环路。为了冗余，两个控制器的后端FC端口必须连接到相同的扩展柜上，所以这台盘阵可以连接的FC-AL的Loop总容量为4个。

其实，磁盘阵列控制器本身就是一个现代计算机系统，它由IO设备、运算器、存储器、软件组成。

- IO设备：包括后端FC适配器、前端FC适配器、管理用COM口、以太网口、LCD液晶显示板、指示灯以及各种适配卡。控制器从后端的磁盘上提取数据，经过虚拟化之后，发送给前端的主机。

控制器从前端的FC端口处接收主机发送的指令和数据，经过去虚拟化运算之后，通过后端FC端口写入扩展柜中的磁盘。这就是控制器工作的基本原理。

- **运算器：**完成上述虚拟化和去虚拟化过程所需要的运算单元。控制器可以选用通用CPU来作为运算器，也可以选用或辅以专用ASIC芯片来完成运算。随着现代盘阵系统虚拟化功能的日益强大，软件逻辑越发复杂起来，通用CPU加软件就成了普遍使用的组合。至于ASIC等硬件只是作为一种IO设备而存在，辅助CPU进行专用逻辑的运算，并把结果返回给CPU，目的是将CPU从这种专用运算中解脱出来。
- **存储器：**包括高速缓存存储器和外部低速永久存储器。现代中高端盘阵几乎都是使用带ECC错误纠正的DDR SDRAM作为高速缓存。这个高速缓存既充当虚拟化软件运行时的空间，又充当两端数据流的缓存空间。
- **软件：**由于虚拟化引擎越来越强大，新的功能和概念层出不穷，单纯使用精简的内核和精简的代码已经远远不能满足功能需求和开发难易度需求。所以目前很多磁盘阵列控制器都是基于某种操作系统内核的，比如Linux、VxWorks、Windows、UNIX等。操作系统不但提供了硬件管理层，还提供了方便的API。这些层次的划分使开发人员只需要专心设计上层虚拟化程序而不是全盘兼顾。盘阵的操作系统和应用程序可以被存放在后端的磁盘上，也可以用专门的外部存储设备存放。NetApp的FAS系列产品就是用一块Flash卡来存放其操作系统和应用程序的。

图9-46的下方是一个扩展柜，其中插了16块FC接口的磁盘。左边控制器的第一个FC接口通过光纤连接到了扩展柜左边的接口（一般为SFP接口），线路将扩展柜内所有16块磁盘串接起来形成了一个Loop。然

而，16块磁盘是远远不够的，怎么在这个Loop中接入更多磁盘呢？肯定是要增加扩展柜的数量，还必须将多个扩展柜中的磁盘都串接到一个Loop上。所以在每个扩展柜的左边接口板上都有两个SFP光纤接口，一个进，一个出。这样就可以把多个扩展柜中的磁盘都串在一起形成一个Loop，接入控制器的一个后端FC接口。

扩展柜右边的接口板与左边构造和连接方法相同，只不过右边的接口需要连接到机头右边的控制器上，形成冗余。这样一旦左边的控制器故障，右边的控制器可以立即接管所有工作。

提示： 扩展柜虽然说通常是一个JBOD，但是随着SBOD技术的普及，扩展柜也变得复杂起来。SBOD技术需要一系列智能芯片。柜子中的磁盘首先要插到一个背板上，背板上提供了一系列的SCA2型母槽。在背板上一定有某种接口来连接这一系列的芯片。通常都是将这些功能芯片单独做到一个模块上，然后将这个模块与背板对接，进而与磁盘接口对接通信。这样，如果为了实现更多的功能，可以只通过更换模块来升级，而不用大动干戈更换整个背板。所以目前几乎所有厂家的盘阵扩展柜都采用这种设计，在柜子后面可以看到两个互为冗余的模块，它们都连接到了同一个磁盘背板上。

9.5.1 IBM DS4800和DS5000控制器架构简析

1.DS4800控制器架构简介

在图9-47中可以看到两个供电模块、两个控制器模块和一个背板模块，两个控制器都连接到了背板上。

图9-47 DS4800机头三维示意图

图9-48是IBM DS4800磁盘阵列的双控制器机头的背面接口图。图中的Drive-side connections代表这些FC端口是用来连接磁盘扩展柜的，Host-side connections代表这些FC端口是用作前端主机的。可以看到上下两个控制器是互为冗余的，它们同时插在机头的背板上，之间有专门的链路进行通信以交互各自的信息。

图9-48 DS4800机头后视图

图9-49是用这台机头挂接16个扩展柜，并且全冗余的架构图。我们可以看到，每个控制器的后端接口都连接了4个扩展柜，这4个扩展柜中的磁盘同时位于主控制器的一个Loop和备用控制的一个Loop上。

图9-49 DS4500连接16个扩展柜

一旦机头上的主控制器发生故障，备份控制器可以立即接管所有工作，继续执行IO请求。因为备份控制器与主控制器一样与所有扩展柜都有连接。

同样，一旦某个扩展柜发生故障，比如电源故障，整个环路从一方来看，是被断开的。但是，其他扩展柜依然可以被访问到，办法就是通过机头上的备用控制器从尾部访问被故障扩展柜隔断的底下的扩展柜，同时主控制器从头部访问上面的控制器。

图9-50所示为每个扩展柜组中都有一个扩展柜故障，但是剩余的扩展柜依然可以继续使用，方法就是让主控制器和备用控制器同时工作。

图9-50 扩展柜整柜故障时的拓扑

可以发现，如果某个Raid Group的磁盘全部在一个扩展柜中，那么一旦这个扩展柜故障，这个Raid Group将不可用。所以控制器为了获得高度可靠性，一般会尽量跨扩展柜做Raid Group，即一个Raid Group中的所有磁盘各属于不同的扩展柜。这样，即使一个扩展柜失效，那么对于一个Raid Group来说，只是失去一块磁盘而不是全部失效，Raid Group还可以继续工作。目前几乎所有中高端盘阵都提供这种支持。

图9-51是DS4800盘阵控制器内部简单架构图。

图9-51 DS4800机头控制器简单架构图

从图中可以看到如下部件：FC Chip、Loop Switch、Channel及Interconnect Module。其中，FC Chip是处理FC协议逻辑的主要芯片，全部的FC逻辑都在此芯片内实现。Loop Switch在这里可以使用上文所述的PBC一类的芯片，当然也可以使用交换架构的芯片。其后端连接两个FC接口，前端分别连接位于两个控制器上的两块FC Chip，这样做的目的是为了充分冗余。Channel的意思就是指连接到一个Loop Switch上的两个FC端口，两个端口组成一个Channel，没有实际物理意义。Interconnect Module其实是一个背板，用于控制器之间的通信。

ASIC芯片负责数据在前端和后端的流动，还负责用于Raid 5/6的XOR运算，由于其XOR运算引擎被固化于ASIC电路中，所以运算速度远快于使用普通X86 CPU。控制器中还有一个Intel Xeon CPU（Control Processor），这个CPU主要用来运行VxWork操作系统，一些上层功能软件比如Remote Mirror、Snapshot（FlashCopy）以及其他总控程序比如监控、用户接口等，皆运行于VxWork操作系统内，由Control Processor负责执行。控制器为ASIC与Control Processor各单独配备了RAM。

推论： 假设某一时刻，左边控制器下方那个FC Chip失效，则控制器A的4号FC接口与这个Chip的通路便会断掉。此时只能通过控制器B的1号FC接口访问最左边的4个扩展柜。查看一下连线，可以发现控制器B的1号FC接口其实是通过Interconnect Module连接到了控制器A的上面的FC Chip。

提示： 所以，最终数据通过控制器B右边的Loop Switch流向Interconnect Module，然后流到控制器A上面的FC Chip。也就是说，最终达到了同一个控制器内部两个FC Chip之间的冗余备份。而此时对前端主机来说并没有影响，主机还是连接控制器A来读写数据。同样，如果控制器A上的某个Loop Switch失效，则数据全部通过控制器B对应的Loop Switch流向Interconnect Module，最后还是流回到控制器A，对主机并没有影响。

但是，如果同一个控制器上的两个FC Chip或两个Loop Switch都失效了，甚至整个控制器故障了，那么所有IO访问就都要转移到另外的一个控制器上。此时，对主机端就会产生影响，需要多路径软件和盘阵端配合参与故障切换的动作。

实际上，DS4800的控制器不一定是上述的切换模式。但是在理论上，所有产品的设计都应该尽量不切换控制器，因为切换控制器会对主机端造成影响。

图9-52为DS4800扩展柜连接示意图。

图9-52 DS4800扩展柜连接通路示意图

图9-53是DS4800控制器的内部详细架构图。

图9-53 DS4800控制器详细架构图

说明如下：

- 2.4 GHz Xeon processor运行Vxworks实时操作系统。
- 每个控制器2、4、8GB数据缓存RAM——数据专用。
- 每个控制器512MB RAM系统缓存——操作系统运行内存。
- XOR ASIC Engine专用芯片，硬件RAID运算引擎。
- 64b/133MHz——1GB/s带宽的PCIX总线。

2. DS5000控制器架构简介

2008年IBM发布了新一代的中端传统磁盘阵列系统DS5000系列。其基本架构设计与DS4000系列大致相同，但是在总线速度和数量方面成倍提升，直接就导致了整体吞吐量飙升。DS4000时代使用PCI-X作为数据IO总线，而且前后端各4个4Gb（1.6GB/s）速率的FC端口各共享一条PCI-X（1GB/s）总线，直接导致瓶颈，系统处理能力受到很大限制。DS5000时代彻底解脱了FC Chip的窘境，每个FC Chip独享一条PCIE 8X（2GB/s）总线直连到核心ASIC，每个FC Chip管控4个4Gb的FC接口，加之每控制器的4个4口无阻塞交换的Loop Switch，这样，系统各处都不存在瓶颈。系统吞吐量的提升，需要更大容量的缓存助阵，所以DS5000系列的缓存容量也比DS4000系列上了一个台阶。

如图9-54所示为DS5000单控制器架构简图。可以与DS4000系列架构简图作一个对比。

图9-54 DS5000控制器示意图

图9-55所示的是DS5000双控制器组成的整体控制架构图和各模块介

绍。

图9-55 各模块介绍

图9-56～图9-58所示为单控制器内部各个模块简介。

图9-56 内部各模块介绍（一）

图9-57 内部各模块介绍（二）

图9-58 SAS的4X宽端口

图9-59所示为DS5000双控制器连接扩展柜拓扑图。图9-60所示为后端接口卡与前端接口卡之间的数据流程。

图9-59 连接扩展柜示意图

图9-60 前后端IO流程

图9-61所示为单个控制器在接收到一个1MB大小的写IO时的处理动作。假设写IO对应的Raid Group中的磁盘都在同一个扩展柜中，而且Raid Group的条带宽度恰好为1MB（不包含Parity Segment），则控制器上的ASIC逻辑电路会自动将这1MB的数据切分成4部分，并且计算好Parity数据。然后将这4块+Parity数据按照奇偶分类分别传送给后端两个FC Chip，其中一个Chip将数据从本地的Loop Switch发出并传送给对应扩展柜写入磁盘，另一个Chip则通过背板上的导线将数据传送给另一个控制器上的Loop Switch，然后发送到同一个扩展柜中对应的磁盘。这样就可以做到充分的负载均衡，这里的负载均衡只是底层硬件芯片处理能力的负载均衡而不是控制器自身整体处理能力的负载均衡，试想，如果将所有Loop Switch放到同一个控制器上，对整个系统性能没有影响。当然，你也可以认为是供电负载均衡了。

图9-61 处理1MB的写IO

如图9-62所示，如果Raid Group中的磁盘不在同一个扩展柜中，那么FC Chip传输数据的路径就会有所变化，会使用多个后端Loop Switch来传输到对应的扩展柜。

图9-62 处理写IO的另一种流程

图9-63所示的是DS5000单个控制器的俯视图。最左侧为专供Control Processor的一条内存。右侧为USB闪存，用于在系统以外掉电时将Data Cache中的内容写入Flash中保存，以待下次启动之后完成未完成的工作。下部的左右两个卡为前端主机接口卡，其上各有一个FC Chip，每个卡上各有4个4Gb/s的FC接口。上部右侧分布在黑色散热片两边的为数据Cache。每条按照2GB算，最大可以支持16GB的Data Cache。所有内存都插在一块单独的插板上，黑色散热片下面覆盖的是内存插板控制芯片。

图9-63 控制器内部俯视图

图9-64所示为将主机接口卡和内存插板拿掉之后露出来的主底板上的部件，两个白色散热片覆盖的为后端磁盘FC Chip芯片。下部为4个整齐排列的Loop Switch芯片。左侧大的白色散热片覆盖的是Control Processor，即Intel Xeon CPU。小一些的白色散热片覆盖的是核心ASIC芯片。没有散热片的大黑色芯片为系统桥芯片。

图9-64 控制器底部俯视图

如图9-65所示为DS5000控制器机柜+扩展柜组成的存储系统整体前视图。

图9-65 整体前视图

9.5.2 NetApp FAS系列磁盘阵列控制器简析

1. FAS2050磁盘阵列

图9-66为FAS2050磁盘阵列控制器的后视图。

从图9-66中可以看到上下两个控制器，每个控制器各有两个FC接口。

图9-66 FAS2050控制器实物后视图

2. FAS3050磁盘阵列

图9-67为FAS3050磁盘阵列控制器后视图。

图9-67 FAS3050单个控制器实物后视图

图9-67为单个FAS3050的控制器机头后视图。图中只是一个控制器，如果要达到完全冗余，可以用两台控制器形成Cluster结构。FAS3050有4个板载FC接口，可以通过插PCIX接口的扩展卡来扩充FC接口的数目。每个控制器提供4个扩展卡槽位，可以插接FC卡、以太网卡、TOE卡和ISCSI卡等扩展卡。

NetApp的FAS产品，从FAS3000系列开始，由于处理功能增强，扩展槽位增多，所以一个控制器就占满一个机头的空间。两台控制器之间需要通过NVRAM卡上的Infiniband网络来形成Cluster。

3. FAS6070磁盘阵列

图9-68为FAS6070磁盘阵列控制器后视图。

图9-68 FAS6070单个控制器实物后视图

FAS6070是NetApp公司比较高端的设备，其内部后端总线的总理论带宽可达32GB/s。有9个扩展槽位，可以插接扩展FC适配器以便连接更多的扩展柜，或TOE卡、以太网卡等其他适配器。

4. DS14MK2FC磁盘扩展柜

图9-69所示为DS14MK2FC磁盘扩展柜的前视图。

图9-70所示为DS14MK2FC磁盘扩展柜的后视图。

图9-69和图9-70所示的是用于连接FAS系列控制器的磁盘扩展柜，每个柜子可以插14块FC接口的磁盘。从后视图中可以看出其与FAS2050控制器的拓扑比较像，不要搞混。前文说过，现代磁盘扩展柜都是用双模块设计，模块中有半交换SBOD芯片。盘柜中所有的磁盘利用其SCA2接口连接到背板上，这个物理接口中所包含的两个逻辑接口各通过电路连接到一个扩展柜控制模块上，形成双Loop冗余。

图9-69 DS14MK2FC磁盘扩展柜前视图

图9-70 DS14MK2FC磁盘扩展柜后视图

9.5.3 IBM DS8000简介

DS8000系列利用两台P系列主机充当控制器。上文中说过，盘阵控制器架构本质上与主机架构无异，DS8000就是这样一个例证。图9-71和图9-72为DS8000实物图。

图9-71 DS8000主机架

图9-72 DS8000盘阵主机架和扩展机架

IBM DS8000磁盘阵列是IBM磁盘阵列产品线中的最高端产品。它利用两台IBM的P系列服务器作为控制器运行AIX操作系统，操作系统之上运行了DS8000的存储虚拟化引擎和管理软件。

9.5.4 富士通ETERNUS DX8000磁盘阵列控制器结构简析

图9-73为富士通ETERNUS DX8000机柜布局示意图。

图9-73 ETERNUS DX8000机柜示意图

与NetApp FAS6070和IBM DS8000系列一样，高端磁盘阵列系统由于需要提供极高的存储容量，只连接几个磁盘扩展柜已经无法满足要求。所以需要将更多的磁盘扩展柜装入扩展机架中，然后统一联入控制器或者串联到其他机架。

图9-74为ETERNUS DX8000控制器架构示意图。

图9-74 ETERNUS DX8000控制器架构示意图

CM（Control Module）为控制器模块。CM是整个磁盘阵列的计算中心，每个CM包含两个2.8GHz的四核CPU和最大64GB的RAM。RAM既作为盘阵本身软件的运行内存，又作为盘阵数据缓存（图中所示的Cache）。整个盘阵系统最多可以安装8个CM，由于每个控制器机柜只包含一个CM，所以整个系统最多可以连接8个Controller Enclosure。

RT（Router）为路由器。RT模块相当于普通服务器架构中的南桥控制器（IO控制器）以及FC架构中常用的FC Loop Switch交换芯片。不同的是这里的RT是一个全局桥，它可以桥接整个系统中的所有CPU、

RAM和外设（FC接口卡），使所有部件之间实现高速通信。每个系统最大可以接入12个RT。关于RT的细节将在下文描述。

DA（Driver Adapter）为后端磁盘通道适配板。每个适配板上接有4个FC-AL接口，用于接入FC-AL的Loop。DA实际上就是盘阵的后端接入点。每个后端RT可以接入16个DA上的各一个口，每个系统最多接入16个DA，从而可以挂接64条Loop。但为了冗余，每个Loop需要同时连接两个DA，所以实际可用Loop减半，为32 Loops。

CA（Channel Adapter）为前端通道适配板。CA实际上就是前端FC接口适配板，每个板子上包含4个FC N类型端口，用于接入FC交换设备或者直接连接主机服务器的FC适配卡。每个RT最多接入4个CA，所以整个系统最多可以接入32个CA，最多提供128个FC协议N类型前端接口。

1. SBOD

扩展柜同样采用了双模块板设计，可以保证两条路径到达同一块磁盘。图9-75中的“骨干交换机”其实就是指SBOD所采用的半交换式芯片。

图9-75 ETERNUS DX8000扩展柜架构

2. 循环镜像的写缓存

图9-76 为ETERNUS DX8000控制器间循环缓存镜像写示意图。

图9-76 ETERNUS DX8000循环缓存镜像写示意图

数据在被写入任何一个控制器的Cache时，系统会将写入的数据复制到其他控制器的缓存中做冗余。一旦在数据还没有写入硬盘之前，某个控制器发生了故障，写缓存镜像技术可以保证数据不会丢失，可以将数据从镜像缓存中写入硬盘。读缓存不需要这种技术，因为读操作只是将数据从硬盘上读入RAM，如果此时控制器故障，磁盘中的数据依然存在。最重要的是缓存镜像技术会浪费宝贵的缓存容量，读操作没有必要实现缓存镜像。

3. 跨扩展柜做Raid Group

图9-77 所示为ETERNUS DX8000跨扩展柜的Raid Group示意图。

图9-77 跨扩展柜的Raid Group

前面也介绍过，为了获得足够的冗余性，很多厂家的盘阵产品一般都选择将不同扩展柜中的硬盘容纳到一个Raid Group中，而不是让一个Raid Group包含同一个扩展柜中所有的磁盘。

DX8000系统整体架构图及分项解析如下。

如图9-78所示为DX8000存储系统配备4个CM时系统的整体架构图。

图9-78 DX8000配备4个CM时系统的架构

图9-78为一个4 CM配置的DX8000存储系统。每个CM包含两个4核心CPU、64GB的RAM、4个前端FC接口卡和两个后端磁盘FC通道接口卡。每个接口卡都含4个FC接口。在IO总线方面，由于所有CM上的DA和CA都需要被系统内所有CPU所访问到，所以通过两个16口8X PCI-E

Bridge Switch芯片来互连所有4个CM，这两个PCI-E Switch作为整个系统的全局IO桥。此外，CM之间的其他数据通信和控制器通信也都经过这两片Switch。由于每个8X PCI-E接口可以提供双向共400MB/s的吞吐量，所以每个Switch芯片可以提供64GB/s的吞吐量。

在后端，DA与磁盘扩展柜之间存在4个16口的FCAL Loop Switch，这个Switch就是前文中所述的Cut Through Switch，物理全交换，但是上层协议依然为FCAL。每个DA上的一个接口各连接4个Switch中的一个Switch上的一个接口，这样，恰好有32个端口被DA所连接。磁盘扩展柜由于也是双路连接，所以每个Loop的头一个磁盘扩展柜也连接到两个Loop Switch上的各一个接口，这样共连接了16条Loop。这种连接方式可以使得每个CM都有若干条冗余的路径来访问到每块磁盘，做到了所有逻辑部件的全部冗余和负载均衡。

如图9-79所示为DX8000存储系统实物图，可以看到CA插板插在CM模块的下方插槽中，CM模块又插到机柜背板中。

图9-79 DX8000实物图

9.5.5 EMC公司Clariion CX/CX3及DMX系列盘阵介绍

1. Clariion CX系列产品

CX系列产品性能如下。

- 充分冗余体系结构。
- 双存储处理器。
- 电源、冷却系统、数据路径、独立电源。
- 无间断操作。

- 在线软件升级。
- 在线硬件更改。
- 高级数据完整性。
- 镜像写缓存。
- 发生电源故障时将写缓存转储到磁盘。
- SNiFFER：扇区检查实用程序。
- 点对点DAE设计。
- 具有无中断故障切换的双I/O通道。
- 分层容量。
- 15K：36 GB、73 GB、146 GB；
- 10K：73 GB、146 GB、300 GB。
- 500 GB SATA II。
- 5~480个磁盘。
- 灵活性。
- 混合驱动器类型。
- 混合RAID级别。
- RAID级别0、1、1+0、3和5。
- 高达16 GB的可调式缓存。

图9-80为CX系列产品的实物图和透视图。

图9-80 CX系列产品实物图和透视图

图9-81为CX700的控制器架构示意图。

图9-81 CX700的控制器架构示意图

图9-82为CX700控制器实物图。

图9-82 CX700控制器实物图

CX3-80控制器和扩展柜架构简介如下。

CX3系列为EMC公司的中低端存储系统（目前已被CX4系列取代）。其内部总体架构与前文介绍的CX700类似，但是在扩展性和总线速度等处有提升。控制器内没有专用XOR芯片，所有数据IO控制全部由主CPU承担，所以整个系统有多个CPU以便提供足够的处理能力。

如图9-83所示为CX3-80单个控制器拆掉挡板后的前视图。

图9-83 CX3-80单控制器前视图

图9-84所示为单控制器后视图，左右两边为两个电源模块。中部上下各一个控制器，每个控制器各包含4个前端FC接口和4个后端FC接口。

图9-84 CX3-80单控制器后视图

图9-85所示为单个控制器的平视图和俯视图。可以看到控制器前面是两个插卡位置，可以根据需要插入对应端口数量的扩展卡。

图9-85 控制器平视图和俯视图

图9-86所示为单个IO扩展卡的平视图和俯视图。可以看到，后端FC接口并没有使用光纤接口，而是使用一种特制的铜缆接口。其上承载的依然是FCAL协议。

图9-86 扩展卡平视图和俯视图

图9-87所示为适用于CX3系列的磁盘扩展柜的前视图。

图9-87 扩展柜前视图

图9-88所示为扩展柜后视图。可以看到中部上下两个可插拔的LCC，即链路控制器，LCC其实就是扩展柜中的控制模块，与IBM DS系列使用的扩展柜中的ESM模块作用完全一样。

图9-88 扩展柜后视图

图9-89所示为单个LCC平视图。用于串联扩展柜的端口有两个，一进一出。

图9-89 LCC平视图

图9-90所示为一套双控制器加5个扩展柜的CX3-80存储系统物理连接好之后的后视图。可以看到，每个控制器有一个后端FC接口使用了一条铜缆，连接到了第一个扩展柜的一个LCC控制模块的入方向级联口上，然后每个LCC的出方向级联口再使用同样的线缆连接到下一个扩展柜相同LCC上的入方向接口，一直级联下去直到最后一个扩展柜为止。

图9-90 控制器+扩展柜后视图

2. Symmetrix DMX-3系统概述

直连矩阵体系结构。

CPU以及通道扩展卡与内存之间采用点对点直连访问。每一个控制器都有其自己到达每一目的地的专用通道。直连矩阵底板最多有128个全部是直连、专用而且不共享的独立通道，如图9-91所示。

图9-91 Symmetrix矩阵示意图

- 每个控制器8个1.3 GHz PPC处理器。
- 最多12个通道控制器。
- 8端口2 GB光纤通道。
- 8端口ESCON。
- 4端口多协议——2 GB FICON、iSCSI和用于RDF的千兆以太网。
- 最多8个磁盘控制器。
- 每个磁盘控制器最多480个驱动器。
- 支持无中断添加控制器。
- 高达512 GB全局内存（256 GB可用）。
- 带有内存保险存储保护的镜像DDR技术。

图9-92所示为Symmetrix DMX-3的实物图，左边为控制器机柜，右边为磁盘扩展柜机柜。

图9-92 Symmetrix DMX-3实物图

9.5.6 HDS公司AMS2000和USP系列盘阵介绍

1. AMS2000系列存储系统控制器架构简介

AMS2000系列是HDS公司于2008年推出的后端基于SAS/SATA磁盘扩展柜的中端存储系统。其控制器有一个很大亮点，即Native Dual Activated。传统双控制器中端存储其创建的每一个LUN，都只能将其指定给一个控制器，另外一个控制作为这个LUN的后备控制器作为故障时的冗余切换，使用多LUN轮流指定给不同控制器的方式做到负载均衡，这种负载均衡完全是人为控制的，如果随着系统的负载不断变化，一旦将来发生某个控制器所掌管的LUN总体负载很大，而另一个控制器管理

的LUN总体负载却很小，那么此时就需要人为控制将负载大的LUN切换到负载小的控制器上，不但属于高危操作而且还影响主机客户端对LUN的访问。AMS 2000的两个控制器彻底颠覆了这种低效率的设计，两个控制器共同掌控所有LUN，不管数据IO从哪个控制器进入，接受IO的控制器都会自行处理对这个LUN的IO，如果一旦发现自己负载高到一定数值而对方控制器负载低到一定数值，那么接受IO的控制器会将这个IO从内部链路传送给对方控制器由对方来处理。通过这种互相协作的处理方式，系统负载达到了彻底的均衡。两个控制器之间使用两条2GB/s的链路进行协作通信与数据传输。两个控制器的写缓存互为镜像以防止一旦某个控制器故障之后另一个控制器可以立即接管所有写IO操作，镜像缓存的第二个目的是为了当发生负载转移时提高转移速度，因为写IO数据已经被镜像所以无须额外的实体数据传送。

与IBM DS 5000系列一样，AMS2000系列也使用一颗专用ASIC来负责大部分数据IO与XOR校验操作。Intel Xeon CPU负责其他上层功能软件的运行。

图9-93所示为AMS 2500存储系统控制器架构示意图。

图9-93 AMS2K控制器

2. USP1100存储系统架构简介

USP系列机器为目前HDS存储产品中最高端的机器。图9-94所示为USP系列机柜实物图。图9-95所示为USP-V系列的虚拟化功能示意图。图9-96所示为USP系列控制器架构逻辑示意图。

图9-94 机柜实物图

图9-95 USP-V系列虚拟化示意图

图9-96 控制器架构示意图

图9-97所示是USP1100存储系统的一些规格说明。

图9-97 USP1100规格说明

图9-98所示是USP1100存储系统控制机柜的物理透视图。

图9-98 USP1100实物图

图9-99所示为Cache Switch模块，对应图9-98所示的逻辑架构图中的Crossbar Switch与Data Cache。Crossbar作为一个无阻塞交换芯片提供所有FC Chip处理器到所有Data Cache之间的高速访问。图9-100所示为各种前端接口导向器插板，用来适配各种前端网络访问方式，包括FC、ESCON、RCON、FICON、NAS等。

图9-99 Cache Switch

图9-100 各种导向器插板

图9-101所示为USP1100存储系统的磁盘扩展机柜透视图。

图9-101 USP1100磁盘扩展柜机柜

9.5.7 HP公司MSA2000和EVA8000存储系统架构简介

1. MSA2000存储系统简介

MSA2000存储系统是HP公司的低端存储系统，不要与HDS公司的AMS2000搞混。前者为Modular Smart Array的简写，后者为Adaptable Modular Storage的简写。MSA2000存储系统后端也是使用SAS/SATA扩展柜，扩展容量有限。前端可以插ISCSI扩展卡或者FC扩展卡。内部架

构为传统简单架构。图9-102和图9-103所示分别为MSA2000控制器架构逻辑简图和透视图。

图9-102 MSA2K控制器逻辑图

图9-103 MSA2K控制器实物图

2. EVA8000系列存储系统架构简介

EVA系列分为EVA4000、EVA6000和EVA8000。EVA全称为“Enterprise Virtual Array”，是HP公司的中端存储系统。其硬件架构有点特别，即将FC控制卡上常见的PBC芯片的功能放到了外部，用一个Loop Switch替代了，控制器与所有扩展柜都接到这个Loop Switch上。

图9-104所示分别为EVA8000控制器逻辑架构简图和控制柜物理拓扑简图。

图9-104 EVA8000控制器逻辑图和实物图

这些Loop Switch芯片，对于其他厂商来讲都是被设计在扩展柜内部的，但是EVA这个产品却把它放到了外部一个单独供电的交换机中，做成星型拓扑了，所有扩展柜都连接到中心的Loop Switch上，其他厂商产品都是柜与柜串联方式的。EVA这种做法有个好处就是数据可以只经过一跳就可以从控制器的IO接口卡上被转发到对应的扩展柜，其他产品则都要经过在柜子与柜子之间进行多级跳（跳数视IO的目标柜子离控制器远近而不同），增加了延迟。另外，这种做法还能够节约Loop Switch芯片的数量，降低成本。

9.5.8 传统磁盘阵列架构总结

虽然目前生产磁盘阵列的厂家众多，产品众多，但是可以把总体的目前所存在的产品总结为以下几种架构。

- 低端X86 PC Server双控制器架构。比如IBM DS3000、EMC Clariion CX/CX3/CX4系列、HP MSA2000系列、NetApp FAS系列。
- 带有辅助专用芯片的低端X86 PC Server双控制器架构。比如IBM DS5000、HDS AMS2000。
- 高端X86 PC Server高扩展性多控制器架构。比如富士通DX8000。
- 高端小型机架构。比如IBM DS8000。
- 高端大型主机架构。比如HDS USP、EMC Symmetrix DMX4。
- X86 Server集群架构。比如EMC Symmetrix Vmax、IBM XIV、3PAR INSERT.

对于最后一种，也就是X86集群架构的存储系统，将会在本书后面的章节详细介绍。这里为何没有将富士通DX8000也归于集群架构的原因是因为其看似多控制器协作，但是在前端，多控制器共享所有DA和CA，只不过分工不同；在后端，所有DA都可以访问所有磁盘扩展柜，后端Loop并不是由某个控制器独享，所以，它仍为一种全局控制器的X86多控制器架构，与双控制器架构的区别仅为一个“多”字。这种架构处于一种折中状态，前端有点集群化的意思，而后端却依然非集群化。

9.6 磁盘阵列配置实践

9.6.1 基于IBM的DS4500盘阵的配置实例

在拿到一台崭新的磁盘阵列之后，必须对其进行相关的配置，才能让其发挥功能。这些配置包括配置LUN，各个LUN的参数，配置LUN映射。这三项配置是使一个盘阵可以用来存取数据的最基本配置。下面就以一台带有两个扩展柜的DS4500磁盘阵列的配置为例，向大家演示一下这些配置的基本步骤。

Storage Manager软件是IBM公司开发的专门针对其DS4000系列盘阵的配置工具，这个工具可以运行在Windows操作系统上，通过以太网与磁盘阵列通信，从而实现配置。

在配置阵列参数之前，我们先来看一下这个阵列将被用于一个什么样的环境。这台阵列是一家小型公司购买的，准备用于存放公司的机密文件、SQL Server数据库文件、静态Web网站文件和内部邮件数据。整体拓扑结构如图9-105所示。

图9-105 某公司IT系统后端拓扑图

其中SQL Server数据库服务器需要3个LUN存放数据，总容量1TB+。邮件和网站各需要一个LUN，大小分别为300GB+和50GB+。FTP服务器需要两个LUN，400GB+。

所有服务器操作系统均为Windows 2003 Enterprise Server。每台服务器上均要安装两块2Gb/s速率的FC HBA卡，而且必须安装多路径软件。

在一台普通PC上安装盘阵随机带的Storage Manager软件，然后将这台PC用双绞线连接到盘阵的以太网口上，将PC的IP地址配置成与盘阵初始IP地址相同的网段。之后，通过软件主界面添加盘阵的IP地址，这样软件就可以通过TCP/IP协议与盘阵进行通信了。

从图9-106中可以看到这套设备共有3个柜子，控制器所在的柜子是0号柜子，也就是图中右边显示的Controller Enclosure，这个柜子中包含两个互为备份冗余的控制器A和B。Driver Enclosure1和Driver Enclosure 2是两个磁盘扩展柜，每个柜子中包含14块物理磁盘。所有磁盘的总容量为2552GB，这在主界面左边栏中也有提示。

图9-106 Storage Manager的主界面

单击按钮可以查看柜子的供电、散热、温度、控制器电池、插卡、SFP模块等硬件部件信息，如图9-107所示。

图9-107 机器环境监控窗口

1. 配置LUN

在主界面中右击Unconfigured Capacity，创建逻辑磁盘（即LUN），如图9-108所示。

图9-108 创建LUN

出现选择主机类型的对话框，选择这个逻辑磁盘将要为何种类型的操作系统使用，这里我们选择Windows 2000/Server 2003 Non-C，即“Windows 2003非集群”类型，如图9-109所示。

图9-109 选择LUN的类型

提示： 有些磁盘阵列会针对不同的操作系统提供不同类型的LUN。虽然LUN对于操作系统来说只是一块裸磁盘，但是每种操作系统上的文件系统在使用LUN的行为上是有差异的。LUN的0号LBA为MBR，占用了一个扇区。而文件系统一般使用比扇区更大的逻辑块来做为分配单元，所以有些文件系统对于LBA的编号就要从LBA1开始，即MBR的下一个LBA。对于一些盘阵控制器来说，它们管理LUN可能也是用块来作为一个最小单元。但控制器却不理解MBR，只认为文件系统会从0开始顺序编号。这样，就产生了块不对齐的现象，从而影响了性能，如图9-100所示。所以，有些以块为单元对LUN做管理的盘阵控制器会针对不同文件系统制作对应的LUN。

图9-110 文件系统与盘阵的LUN块不对齐示意图

在图9-111所示的对话框中，需要先创建RAID Group（或Arrays），然后在建好的RG中再划分LUN。数据库对于IO性能有较高要求，所以给数据库的3个LUN各分配一个RG，充分保证数据库的IO性能。

图9-111 RAID类型选择，手动分配磁盘

选择相应的RAID级别。如果想手动选择组成RAID Group的磁盘，选中Manual单选按钮。然后在下方磁盘列表中，按住Ctrl键选择组成这个RG的所有磁盘成员，最后单击Apply按钮，如图9-111所示。如果想让程序自动生成各种大小的RG，则要选中Automatic单选按钮，如图9-112所示。

图9-112 自动分配磁盘

这里让系统自动为我们计算。在RAID 5级别下，系统计算出各种容量、磁盘数量的RG组合。Channel Protection指的是让RG中所有磁盘成员分布在两个扩展柜中，充分保证冗余性。我们选择5块盘的292GB容量。

在这个RG中，我们可以手动指定大小，划分出新的LUN（即Logical Drive）。这里将全部容量分配给一个LUN，命名为SQLLUN1，LUN的参数选择自定义。

选择条带深度为32KB，每个条带的数据部分的宽度就是32KB乘以4块数据Segment，为128KB。这样，只要将SQL Server数据库的Extent参数设置为128KB大小，就可以保证每次读写RG均为整条读写，充分提高IO性能（但这样做会丧失并发IO能力），如图9-113所示。Preferred controller ownership表示这个RAID Group平时由哪个控制器管理。我们选择Slot A，即控制器A。一旦控制器A发生故障，这个RG会立即由控制器B接管。

图9-113 选择条带深度

Logical Drive-to-LUN Mapping即LUN映射，配置这个LUN将给哪个或者哪些主机使用。我们稍后选择用专门的配置模块配置，如图9-114所示。

图9-114 配置LUN映射

配置完后，系统会提示是否还要配置另外的RG或者LUN。因为这个RG全部容量已经分配给了一个LUN，所以窗口中的same array单选按

钮为灰色。单击Yes按钮进入新一轮的RG和LUN配置。用这种方法，配置所有6个LUN，如图9-115所示。

图9-115 创建完毕的6个LUN

由于这个环境中的FTP服务器主要存放视频等大文件，所以使用RAID 3级别来提高传输性能，如图9-115所示。另外，只要选中某个创建好的RG或者LUN，右边窗口中便会显示出这个RG和LUN所对应的物理磁盘以及掌管它的控制器。

2. 设置全局热备磁盘

最好设置一个或者两个热备磁盘。这样，一旦某块磁盘损坏，热备磁盘立即顶替，以免数据丢失，如图9-116所示。

图9-116 选择热备盘

设置完热备磁盘之后，可以进一步更改每个RG或者LUN的细节参数，如图9-117和图9-118所示。

图9-117 各种参数（1）

图9-118 各种参数（2）

3. 设置LUN映射

切换到主界面的Mappings View选项卡，然后右击设备名，从弹出的快捷菜单中选择Define Host命令来添加主机信息，即使用这台盘阵的主机信息，如图9-119所示。

图9-119 添加主机映射

Host Name可以随便起名，并不一定要与对应主机的真正Hostname相同。添加完所有主机后，界面的显示如图9-120所示。

图9-120 主机列表

添加完主机名之后，还必须添加Host Port，即必须让盘阵知道主机使用哪些FC接口来访问盘阵。右击某个主机名，从弹出的快捷菜单中选择Define Host Port命令，如图9-121所示。

图9-121 添加主机的FC端口信息

在Port Identifier处，我们选择对应的ID（即每个主机FC HBA卡上的WWN地址，盘阵会通过Fabric网络自动发现这些地址）。主机类型当然选择Windows 2000/Server 2003 Non-C，Host port name随便起一个名字即可，如图9-122所示。每台服务器上有两块FC卡，所以每台主机共有两个接口，在此为每个接口都定义一下。定义好每台主机的接口之后，如图9-123所示。

图9-122 主机上的FC端口信息

图9-123 所有主机端口定义完毕

在未映射的LUN上右击，从弹出的快捷菜单中选择Define Additional Mapping命令，打开如图9-124所示的对话框。

图9-124 映射LUN到主机端口

在Host group or host下拉列表框中选择Host SQLserver这台主机，Logic unit number表示此次映射给这台主机的LUN，在主机上将显示LUN的号码。图9-124中，将SQLLUN1这个LUN映射给主机SQL Server，主机上显示的对应这块LUN的号码是LUN0。用这种方法，将

所有属于SQL Server主机的3个LUN都映射好后，单击close按钮。结果如图9-125所示。

图9-125 LUN映射完毕

注意：单击每台主机，右边就会出现这台主机可以访问的LUN及其相关信息。每个LUN只能映射给一台主机。理论上，一个LUN完全可以映射给多台主机共同访问使用，这就需要用到partition功能了。此处不做过多介绍。

4. 初始化LUN配置

所有设置做完之后，需要进行初始化。初始化完毕之后，才能供主机使用。

右击每个Array或者LUN，然后从弹出的快捷菜单中选择Initialize命令，如图9-126和图9-127所示。

图9-126 初始化磁盘阵列

图9-127 初始化所有RAID组

右击每个Array，从弹出的快捷菜单中选择属性命令，可以查看初始化的进度，如图9-128所示。

图9-128 初始化进度窗口

初始化完成之后，盘阵即可接受主机访问了。

其他不同品牌型号的磁盘阵列产品的配置过程大同小异，只要理解了磁盘阵列的组成架构和原理，配置起来其实是很简单的。

9.6.2 基于EMC的CX700磁盘阵列配置实例

在此我们简要介绍一下EMC针对CX系列的配置工具Navisphere。

1. 登录Navisphere，查看全局情况

全局视图如图9-129所示。图9-130为所连接的两个扩展柜中的一个柜子里的磁盘列表。图9-131所示为控制器柜里包含的两个控制器的信息。图9-132所示为存放盘阵操作系统及重要配置信息的私有LUN。每个控制器都有各自的私有LUN。图9-133所示为映射LUN与主机用的Storage Groups，也就是映射组。在这个组中的主机可以访问这个组中的LUN。

图9-129 全局视图

图9-130 扩展柜中的磁盘列表

图9-131 控制器柜中的两个控制器

图9-132 用于存放盘阵的操作系统以及其他配置信息的Private LUNs

图9-133 用来映射LUN和主机的Storage Groups

2. 创建RAID组

(1) 在盘阵图标上右击，从弹出的快捷菜单中选择Create RAID Group命令，如图9-134所示。

图9-134 创建RAID组

(2) 在如图9-135所示的对话框中选择RAID组的ID、RAID组所包含的磁盘数量以及其他参数。

图9-135 RAID组参数

(3) 在创建好的新RAID组上右击，从弹出的快捷菜单中选择Bind LUN命令创建LUN，如图9-136所示。在打开的对话框中，我们选择RAID类型为RAID 5，如图9-137和图9-138所示。利用上述方法创建LUN 6和LUN 7两个LUN，如图9-139所示。

图9-136 在RAID组上创建（绑定）LUN

图9-137 LUN参数页（1）

图9-138 LUN参数页（2）

图9-139 创建好的两个LUN：LUN 6和LUN 7

3. 创建Storage Group并绑定主机与LUN

(1) 在盘阵图标上右击，从弹出的快捷菜单中选择Create Storage Group命令，如图9-140所示。

图9-140 创建Storage Group

(2) 在打开的对话框中输入新Storage Group的名称，如图9-141所示。

图9-141 给新Storage Group起名

(3) 在创建好的Storage Group上右击，从弹出的快捷菜单中选择Select LUN命令绑定LUN，如图9-142所示。

图9-142 选择要映射的LUN（一）

(4) 将要分配给这个组的LUN移动到右边的窗口，如图9-143所示。

图9-143 选择要映射的LUN（二）

（5）切换到Hosts选项卡，选择要分配的主机，将其移动到右侧窗口，如图9-144所示。

图9-144 选择要绑定的主机

（6）添加完LUN和主机之后，对应的主机就可以识别到对应的LUN，并可以使用了，如图9-145所示。

图9-145 创建好的Storage Group

各种磁盘阵列的配置大同小异，总地来说主要有以下三步。

- （1）配置RAID组。
- （2）在RAID组上划分LUN。
- （3）将LUN映射给相应的主机使用。

9.7 HBA卡逻辑架构详析与SAN Boot示例

9.7.1 HBA卡逻辑架构

关于IOP

IOP全称IO Processor，20多年前由IBM首先提出。其根本目的是在计算机IO设备上放置一个智能处理器，来处理一些通信协议的逻辑，从而将原本需要由系统主CPU执行的功能（设备驱动层面）Offload到IO设备自身执行以便节约主CPU资源。时至今日，IOP已经被普遍使用在各种IO设备上，比如以太网卡、FC卡、Raid卡、SAS卡、SCSI卡等。当然仍有一部分卡不使用IOP，因为目前的主CPU性能已经非常强劲，与20多年前的CPU效率不可同日而语，所以这类没有IOP的IO设备，其上层逻辑功能主要由系统CPU完成。当然，随着集成度的提高，各种最新的标准IO板卡基本都是单芯片设计了，IOP也被集成在了芯片内部，而且往往只占很小一部分，其他都是各种接口逻辑、加速逻辑。

带IOP的IO板卡一般都需要一个OS（即Firmware），这个OS代码存放于IO板卡的Flash或者ROM介质中，有些过于庞大的OS甚至干脆存放在与这个IO卡相连接的磁盘特定区域中。有些板卡逻辑较为简单，也可以不使用OS，而直接使用非常简化的单任务Firmware。

不管是IOP或者非IOP的IO板卡，如果需要从这张卡所连接的外部设备启动系统，则都需要一个Optional ROM。Optional ROM的作用见下文。

非IOP的HBA比如Adaptec 2940、LSI 896。

关于**Firmware**

Firmware指运行在IO设备自身的程序代码。其包含Tiny BIOS和OS或者简化的单任务代码。Tiny BIOS用于带IOP的IO卡加电时自身的启动的初始化操作，其包括了各种级别的Boot Loader。OS则用于IOP载入执行以实现这个IO设备自身的功能，比如Raid功能。Boot Loader将OS以及用于其他硬件逻辑的代码加载到对应的内存空间，供这些部件执行。

Tiny BIOS不等于Optional ROM，不要将二者混淆。前者是IO设备上的IOP自己执行的，用于初始化自己；后者则是主板上的主CPU通过系统BIOS载入执行的用于系统从设备外部访问这个IO设备。

关于**Optional ROM**

作用：主要用于Boot。Optional ROM代码可以存放在IO卡的ROM里，也可以存放在主板系统BIOS中（如果你的板卡已经是非常普遍的大众型号的话）。从这个设备Boot之前需要加载这个设备对应的Optional ROM，这样系统BIOS才可以驱动这块卡从而向这块卡后面的磁盘读写数据。也就是说，Optional ROM里包含了这块板卡针对主板BIOS的驱动程序。

IO设备自身可以控制是否在系统加电的时候让系统BIOS加载Optional ROM。系统BIOS必须运行对应外设的Optional ROM才可以输出这个外设更细节的信息，而且也只有运行Optional ROM才能让用户进入这个外设的配置界面进行底层的配置，比如大家经常会看到服务器启动的时候提示按Ctrl+A进入某配置界面。Optional ROM执行完之后，系统BIOS便跳转到下一个存在的Optional ROM继续执行，如果上一个执行过的Optional ROM未进行任何配置变更，则整个系统就未作任何更

改。如果想让系统BIOS访问IO卡后面的磁盘，对于某些卡，仅仅执行Optional ROM是不行的，还需要进入配置界面来打开一个选项，一般叫做“Enable BIOS”。其实这个选项实际控制着是否让这个IO卡设备注册到系统Int13中断向量表中，如果不打开这个选项，则IO卡不向Int13注册，则系统就无法读写IO卡后面的磁盘。下文的配置示例中会给出相关的过程。

HBA架构方框图

如图9-146所示，IOP即为这个IO设备的CPU；RAM为执行代码必需的内存；ASIC则是辅助运算的专用芯片，比如XOR运算芯片，可以大大降低IOP的负担，提高运算速度；Flash用于存放Firmware和Optional ROM；Bus Controller Chip用于总线数据传输的控制，比如PCIE总线控制。

图9-146 HBA结构方框图

非Boot的HBA卡访问流程

不支持Boot功能的HBA卡可以没有Optional ROM。这种HBA卡在操作系统未加载之前不能对其所连接的物理设备进行访问。只有在操作系统加载之后，在操作系统中安装这个设备的驱动程序，方可对其所连接的设备进行访问。

关于操作系统安装与启动

支持Boot的HBA，虽然其Optional ROM可以在OS未加载前提供对其自身的访问，但是由于Optional Rom所提供的访问方式也只是非常低效率的Int13方式，加之传统BIOS只运行在16b模式下，非常古老，根本

不能满足需求（EFI BIOS运行在64b模式下并且对驱动支持方面有了很大改进，另当别论）。所以现代操作系统都要求操作系统内必须包含这个IO卡的高级驱动，除了实现少数一些功能之外，几乎不再调用系统BIOS来访问IO设备。所以，在安装操作系统到某个HBA所挂的磁盘时，要么操作系统安装介质已经包含对应设备的驱动程序，要么就需要手动用软盘等外部介质加载对应的驱动程序。如果HBA所挂的磁盘或者LUN上已经安装有操作系统并且携带了对应的驱动程序，则可以直接从这个LUN启动操作系统。

9.7.2 支持Boot的HBA卡访问流程

具体过程见下文描述。

提示： 一台计算机包含有多个BIOS。主板BIOS，也叫系统BIOS，包含用于访问所有常见IO设备的基本的驱动程序，比如键盘、软驱、光驱、IDE硬盘控制器、USB控制器（一些早期的主板BIOS不包含）等。对于一些不常见也不常用的IO板卡，或者功能比较复杂的IO板卡，比如SCSI卡、Raid卡、网卡、显卡等，这些IO设备一般都具有自己的BIOS，也就是将驱动自己的代码放到自己的BIOS中，这些BIOS会在系统BIOS执行之后逐个加载以便让系统BIOS可以访问这些设备。这些IO设备可以是插卡形式的，也可以是集成在主板上的。它们自己的BIOS可以存放在IO板卡的芯片中，也可以作为附加的代码部分存放在系统BIOS所在的芯片中的附加部分，所以这些BIOS可以被单独升级。这些IO板卡独立提供的BIOS又称为Optional ROM。

Optional ROM对于一个IO板卡来说并不是必需的，只有以下两

种情形才是必需的。一是：在操作系统加载之前需要被访问（比如显卡），则一定需要Optional ROM。二是：功能过于复杂，系统BIOS不包含它的基本驱动代码。DOS操作系统以及基于DOS的早期的操作系统必须依附系统BIOS来对外设进行IO操作，原因就是DOS操作系统自身并不包含任何设备驱动程序，所以只能调用系统BIOS执行期间在内存驻留的所有IO设备驱动程序。但是近代操作系统都会使用自身的驱动程序直接操作IO设备，并不需要系统BIOS的参与了，除非操作系统需要对这些IO设备进行一些非常底层的操作，比如设备重新初始化等。所以，近代操作系统在启动之后，会用自己的驱动程序替代系统BIOS加载的基本驱动程序。

系统BIOS在执行的过程中，会逐个加载所有IO设备的Optional ROM。首先系统BIOS会对实内存地址0xC0000到0xF0000进行扫描，扫描单位是以2KB为边界。所有具有Optional ROM的IO设备均会在内存对应地址中映射这些ROM。系统BIOS一旦发现0xAA55这样的字串，便知道这是某个IO设备的ROM入口，0xAA55便是Optional ROM的签名。在这个签名之后是一个字节的指针，这个指针给出了这个Optional ROM在内存中占据了多少512B的Block。这个指针之后紧接着的内存部分就是ROM实际代码的入口，系统BIOS直接跳转到这个入口开始读取Optional ROM代码执行，Optional ROM会首先向系统BIOS注册其中断向量以便被其他程序所访问，其次Optional ROM还会提供一个用户接口用于配置这个IO设备，比如很多HBA在加电启动之后会在屏幕上显示“Press Ctrl+Q to enter configuration utility”。其次还可以提供对这个IO设备的一些底层诊断。如果某个IO设备支持从其所连接的外部存储介质来启动操作系统，则如果这个设备是非常普遍的比如IDE控制器，那么系统

BIOS中会包含它的基本驱动，则不需要任何其他过程，就可以在系统BIOS的配置界面中选择从这个IO设备启动。但是如果这个IO设备需要Optional ROM，则在未执行Optional ROM之前，系统BIOS中不会识别到这个IO设备，从而也就不会提供从这个设备启动的选项。Optional ROM执行的过程中，会利用一种由系统BIOS提供的通用的API，即BIOS Boot Specification（BBS） API来向系统BIOS注册自身，这样，在系统BIOS配置界面的Boot sequence设置中就可以出现对应的设备和这个设备已经识别到的外部介质了。

仔细的读者可以发现，一般在进入系统BIOS配置界面之前，系统BIOS都会首先完成所有Optional ROM的执行加载过程之后才会进入配置界面，其目的就是为了收集所有可供启动的外设列表。如果在没有加载全部可能的Optional ROM之前就进入配置界面，则用户也不会看到对应的设备，就会造成不便。

（1）我们以Qlogic公司QLA2340 FC卡来举例。开机后按照相关提示进入FC卡的BIOS配置界面。如图9-147和图9-148所示，选择需要配置的卡，进入之后，选择Configuration Settings选项。

图9-147 选择待配置的卡

图9-148 进入 Configuration Settings

（2）如图9-149所示，进入之后，选择Adapter Settings选项，出现如图9-150所示的窗口。

图9-149 选择Adapter Settings

图9-150 Enable BIOS

（3）图中显示的Host Adapter BIOS就是指Optional ROM，将其

Enable，表示允许系统BIOS在执行的过程中加载。退出，然后单击图9-149所示窗口中的Selectable Boot Settings来选择可供启动的LUN，出现如图9-151所示的窗口，此处，选择4个列表中的第一个，系统将列出所识别到的FC Target，如图9-152所示。选择对应的Target，退出。

图9-151 进入Selectable Boot

图9-152 选择识别到的FC Target

（4）出现如图9-153所示的窗口，这个窗口列出了这个FC Target中所包含的所有LUN和其对应的ID。此处，选择LUN0，退出，发现所选择的LUN已经出现在Boot List中，如图9-154所示。

图9-153 进入Selectable Boot

图9-154 选择识别到的FC Target

（5）在配置完成后，退出FC卡BIOS配置界面并重启系统，让Optional ROM重新向系统BIOS注册启动LUN并且加载Optional ROM。这样便可以从LUN启动操作系统了，如果操作系统尚未安装，则需要通过安装盘启动后，加载对应的驱动程序，将系统安装在这个LUN中，如图9-155和图9-156所示。

图9-155 进入Selectable Boot

图9-156 选择识别到的FC Target

9.8 国产中高端FC磁盘阵列

提到Infortrend这个名字，圈内人可谓无人不知无人不晓，但是圈外人可能知道的不多，因为他们的产品几乎被所有二三线厂商OEM过，在国内是可以与经典的LSI贴牌产品平起平坐的。这类产品都有个特点，就是中规中矩、不高不低，该有的都有，没有的可二次开发，硬件品质和做工优良，稳定可靠，特别适合于OEM给那些把持着市场和渠道能力但是却苦于没有产品又不想投入太多研发成本的厂商。Infortrend也算是在存储界打拼了20年的老厂商了，其产品也不知被多少其他后入存储界的厂商研究、学习和借鉴过，其在中国国家专利局有40多项专利。

如图9-157所示为Infortrend的产品Portfolio，其中囊括了同时支持块和文件访问方式的统一存储产品和ESDS系列块存储产品。

图9-157 Infortrend产品系列一览

提示： ESDS4000和ESDS5000系列在落笔时即将上市。ESVA系列属于高端存储系统。我们将会在本书NAS一章里介绍Infortrend的本书存储产品，在集群一章中介绍ESVA特殊的集群方式，在容灾一章介绍Infortrend的容灾复制技术。

Infortrend的FC阵列产品分为两个系列，一个是中低端EonStor DS（ESDS）系列；另一个是中高端ESVA（Enterprise Scalable Virtualized Architecture）系列。

9.8.1 Infortrend中低端ESDS系列存储系统

如图9-158所示为ESDS系列的几种规格，其中包括2U 2.5寸盘24盘位双控、2U 3.5寸盘12盘位双控、3U 3.5寸盘16盘位双控、4U 3.5寸盘24盘位双控，以及高密度的4U 3.5寸盘48/60盘位双控。

图9-158 ESDS3000各种规格的控制柜

市场上能做高密度盘柜的厂商不太多，其中不太好掌握的几个技术是散热和在线热插拔。在有些解决方案中热插拔必须把上机箱盖全部打开，并且要么抽出整个机箱（基本不可行，太沉），要么机箱上方留有足够空间，这种方案在维护的时候很不方便。而另外一些方案将磁盘分为多个组，每个组放在一个抽屉里，可以以抽屉为单元整个拉出，然后更换其中的磁盘。整个抽屉拉出的时候，并不会影响该抽屉内的磁盘IO访问，这就需要抽屉与控制器上的SAS通道芯片/卡之间仍有物理连接。一般是通过一种叫做“坦克链”的形态实现的，或者也可以称之为脐带。如图9-159所示为坦克链实物图，这种设计能保护和稳定SAS和电源连线，并且随着抽屉的推入可以收缩到原有空间内，非常方便。只需要注意抽出和插入时的振动控制即可。

图9-159 坦克链实物图

让人比较惊讶的一点是，Infortrend能够在4U空间内容纳60块3.5寸磁盘+2个存储控制器，这在业界是领先的。如图9-160所示为各种规格控制器机头的后视图。扩展性方面，每个控制器均提供一个可更换的通道模块位。最大可连接5个磁盘扩展柜，如果用60盘位的高密度盘柜，再加上控制器自带的60盘位，总共可以接入360盘。

图9-160 ESDS3000系列控制器后视图

早期的外置存储系统都是使用电池直接保护所有数据缓存，电池电

量有限，如果长时间无法恢复供电，数据就会丢失。最新的设计一般都是使用Flash介质在掉电之后永久存储数据缓存内的脏数据。好处是电池可以不用很大。图9-161左图为ESDS3000的备电模块，包含电池和一块Flash卡，右图则为传统备电方式。

图9-161 ESDS3000备电模块

ESDS3000支持主流的存储协议，包括8G/16G FC、1/10GbE Eth、6Gb SAS，并且提供了各种通道扩展模块，如图9-162所示。

图9-162 ESDS3000控制器扩展通道模块

9.8.2 Infortrend中高端ESVA系列存储系统

ESVA全称是Enterprise Scalable Virtualized Architecture，主打Scale-Out和虚拟化。最大单系统可以Scale-Out到12个节点，每节点400块盘，共4800盘，如图9-163所示。

图9-163 ESVA存储系统

提示： 我们会在第15章讲集群的时候介绍ESVA的Scale-Out和虚拟化方式。

如图9-164所示为ESVA系统内单个节点配置，机头双控挂了一批扩展柜，每个控制器提供一个MiniSAS后端接口，可以串联多个扩展柜，前端则提供最大8个FC接口和4个Eth接口。

图9-164 ESVA单节点示意图

9.8.3 Infortrend存储软件特性及配置界面

在软件特性方面，ESDS和ESVA系列支持主流技术，比如快照克隆、自动精简配置Thin Provision、远程复制、自动存储分级。对于重删这种非主流特性，一般离线或者近线存储用得较多，不支持。对于快照，ESDS支持4096个，ESVA支持16000个；对于自动存储分级，ESDS支持2个层级，而ESVA支持4个层级。另外，ESVA还支持针对Thin卷进行NTFS/EXT2/EXT3文件系统的空间回收，这一点难能可贵。

提示： 我们会在容灾的章节中介绍Infortrend远程复制技术的一些特点。

串口是比较常见的配置接口，多数产品在串口配置方面做得是比较粗枝大叶的，有些只提供命令行接口，人机交换很不友好。令人欣慰的是Infortrend提供了菜单模式的配置方式，极大增强了便捷性，这一点挺难得的，如图9-165所示。

图9-165 串口菜单配置方式

同时，Infortrend的SANWatch GUI图形界面，其实是Infortrend的一个软件套件，其包含了Storage Manager、Replication Manager、Virtualization Manager、Disk Performance Monitor、EonPath多路径软件以及应用快照一致性代理组件。

图9-166 SANWatch套件

SANWatch可以让用户用HTTP方式通过Web页面来配置存储系统，如图9-167～图9-169所示。可以看到其显示的元素还是很多样化的，更加注重用户体验，在如今互联网大潮下，还是能够跟得上时代的。

图9-167 GUI图形界面（1）

图9-168 GUI图形界面（2）

图9-169 GUI图形界面（3）

9.9 小结

磁盘阵列的前端和后端用FC网络作为通路取代原来的并行SCSI通路技术，获得了极大的成功！如果说将主机通道替换成FC通路只能称为半网络的话，那么将一个个磁盘作为FC网络上的节点对待，可就是彻彻底底的网络化存储系统了。这也自然阐释了“网络存储”和“存储网络”的概念。

提示： 此时，我们再来看本书第2章中所描述的那个“网中有网”的模型，可以看到，整个系统，所有通路，都可以说是网络化的了，只不过CPU 内存总线和主机IO总线应该算是半网络化，只是一个总线，上面没有网络化所特有的“协议”和“开销”、包交换等词汇。

目前，NGIO（Next Generation IO）已经提上日程。这种架构就是将主机IO总线甚至内存总线都用交换式网络来连接。可想而知，这个交换网络速率肯定很高，稳定性和可扩展性也很强。

到了Next Generation时代，内存、CPU和各种IO设备可以在地理上相隔很远，甚至可以通过网络共享。内存可以不仅仅被一个CPU使用，多个CPU、多个内存、多个IO控制器之间都是点对点交换式互联通信。这是一个很有吸引力的课题，就像存储网络化一样。对于存储网络化，位于A地的主机可以识别并使用远在相隔千里的B地一台盘阵上的LUN。那么对于网络化的系统总线，A地的CPU可以访问位于B地的内存阵列或者位于B地的某个IO控制器，它们之间都通过网络相连。

说明： 有人将上面描述的架构，称为System Area Network，即

SAN（系统区域网络）。有意思的是，它和Storage Area Network的简称同名。

目前基于PCI-E网络的MR-IOV已经逐步形成标准，包括Infiniband、NGIO、FutureIO在内的多种太过超前的“将计算机总线长出主机外面”的想法，现在看来都没有成为主流。那么这次依赖开放式低成本高速的PCI-E通道技术而生的MR-IOV，同样也是这个目的，它是否能够发展为主流并促成革命，让我们拭目以待！

第10章 三足鼎立——DAS、SAN和NAS

■ DAS

■ SAN

■ NAS

FC已经成功地将传统的磁盘阵列改造成了彻底网络化传输的磁盘阵列，不仅从盘阵到主机的通路成了网络化，就连盘阵后端控制器到磁盘的连接也被彻底网络化了。尤其对盘阵后端的改革更是一个惊人的创举！

盘阵后端的网络化使可接入磁盘节点数大大增加，可扩展性大大增强。一时间各个厂家纷纷制造出自己的盘阵，由于后端接入容量增加，这些盘阵不是几个磁盘箱就能放下的了，它们动辄就要占用几个大机柜。机房中占地最大的往往就是存储设备，而不是主机设备。

存储区域网络（Storage Area Network, SAN）这个概念，直到FC革命成功之后，其意义才真正体现出来，存储才真正走向了网络化。在广义中，各种存储架构都可以称为SAN，因为就算直接连到主板上的IDE通道也可以连接两块磁盘。从这种意义上说，它就是一个2节点网络。

10.1 NAS也疯狂

武当乃张真人所创。想当年，张真人就是在武当创立了卷管理和文件系统的伟大理论！如今，张真人已经逝去多年。然而，他的理论却被广泛地应用着，而且深入人心。不过，没人会追究起到底是谁创立了这些理论，因为它太广泛了，广泛地以至于没有人去理会它了。

10.1.1 另辟蹊径——乱谈NAS的起家

武当也跟上了时代的变化，不但本身的体制从公有制改成了股份制，而且董事会还决定将武当仓库全面对外开放，利用一切可以利用的手段盈利。除去货币贬值的因素，收费比张真人时代贵出了好几倍。利润大部分属于武当董事长瓜董，剩下的除了给员工开点工资外，全部用来扩容仓库和加强仓库建设，以获取更大的利润。

1. 武当仓库简要介绍

武当仓库是一个历史悠久、源远流长的大型仓库，其创始人是张真人。起初只是为了满足武当本派存放货物使用，后来对外开放。仓库分为8个库区，每个库区中又有不计其数的房间。整个仓库配备了两名仓库管理员，各自管理4个库区。一旦某个管理员请假不能到岗，另外一位管理员就要暂时管理全部库区。仓库共有东西两道大门，各由一位库管员把守。仓库地理位置优越，其前方就是一个立交桥大枢纽系统，当地路政部门给予武当极其优越的条件，专门为武当仓库的两道大门修了能直通大枢纽的高速路。

武当仓库当前的运作模式如图10-1和图10-2所示。

图10-1 武当仓库当前运作模式

图10-2 当前后端存储网络架构

自从武当仓库宣布对外全面开放后，短短的一天内，8个库区全部被预售一空。买家当然都是赫赫有名的大门派，因为也只有他们才出得起高价。

由于体改的时候，当年被张真人看好的微软道长被瓜董打发回家了，所以现在武当仓库只有两位根据提货单和入库单进行取货、存货的库管员。

当前武当提货单/存货单格式如图10-3所示。

图10-3 当前武当仓库货单

每个门派都通过交通枢纽来向武当仓库存取货物。武当存取货物的单据上没有对货物属性的描述，仓库管理员只需根据货物存取单据上描述的房间号段来将对应的货物取出或者存入，而不管这些货物是什么，有多少。

所以，每个门派要有一个理货员，这个理货员知道什么货应该去哪些房间提取，以及有多少个空闲房间。他自己保留了一个房间使用情况图，每当本门派需要提取什么货物，理货员就根据这个对应图计算出货物对应的存储房间，然后填写武当仓库货物存取单交给仓库管理员，管理员将对应房间中的所有货物交给理货员，理货员在将货物整理好后交给本门派使用。存货的过程也类似，理货员记录好货物要存入的房间，然后填写货物存取单，将存取单和货物交给仓库管理员。管理员根据房间的号码将货物依次放入对应房间。值得说明的是，货物存取单上的房间必须是连续的，不允许断开存放。连续的房间数量最大是128个，超

过128个房间的货物，就需要填写多张货物存取单了，如图10-4所示。

图10-4 填好的货单

武当瓜董天天研究着怎么从现有的资源中，榨取最后一滴利润。经过调查，他发现各门派都养着一名理货员。瓜董心想其他门派一定也很头疼，能少养人就少养人。这天晚上，反复琢磨，他终于想到一个绝招。由于各门派目前都养着一名理货员，他们要付劳务费。如果用武当的人来充当理货员，卖服务给各门派。而每个门派都会付一份劳务费给我，而我只需要付一份工资给理货员就可以了。真是一本万利啊！

另外，由于江湖政府换届时，瓜董没有搞好关系，使武当每月需要向江湖政府缴纳高额的高速公路使用费，这让一向以节省成本著称的瓜董苦不堪言。瓜董决定抛弃高速公路，使用普通公路。

值得称赞的是，瓜董不是个爱面子的人，他为了节省成本，可以不惜一切代价。他亲自把微软道长请回了武当，让他担任仓库理货大总管职务。随后，瓜董联系了各门各派掌门人，让他们把理货的工作外包给武当做，劳务费比原来有所优惠。各门派都同意了。

微软道长手下可以有多个分管。各个门派使用的货物记录方式并不相同，大部分门派使用的是微软道长所创立的NTFS记录方式，而有些则使用的是少林雷牛大师所创的EXT格式。既然要将货物记录服务全部外包给武当，那么微软道长无疑要将所有这些门派使用的记录方式都实现，从而为每个门派服务。微软道长遂发布告示，广招天下贤士来任职分管职务，每个分管管理其各自的货物记录方式。图10-5和图10-6为改制之后的仓库管理模式。

图10-5 改制之后的仓库架构

图10-6 改制之后的后端存储网络架构

2. 改制之后的仓库架构

武当仓库经过这样的改造之后，功能更加强大了。唯一不足的是为了节省成本将原本高成本的高速公路替换成了普通公路。

原来的单据显然不适合仓库当前的运作模式了，仓库的货物存取单也改版了，如图10-7所示。

图10-7 改制之后的货单

从武当仓库整改之后的新单据中可以发现，现在各个门派只要告诉武当仓库的理货员需要什么货物及多少数量就可以了，完全不用记录这些货物实际放到了哪个房间，在哪里及怎么去这些内容。计算这些复杂的对应关系的工作完全移交给了武当仓库理货员来做。理货员计算好之后，生成提货/存货单，交给库管员，直接去对应房间提/存货物。理货员与库管员之间的交互，速度快了很多，因为完全是在仓库内部进行通信了，不需要跨越缓慢的公路交通系统。

经过实践，瓜董的这套做法还真取得了显著成效，各门派无须雇用理货员，无须支付昂贵的高速费，节省了成本。同时，瓜董也发了财。不过唯一不足的是对于普通公路进行货运的速度，各门派不太满意。但是相对于成本大大降低和便捷带来的好处，各门派也只有牺牲速度了。

10.1.2 双管齐下——两种方式访问的后端存储网络

但是有的仓库租用者对这种方案并没有兴趣，他们不但追求速度，而且情愿用自己的理货员，也不相信仓库提供的理货员。这种客户得罪不起，那么就给他单独的政策，还是采用原来的方式使用仓库，和新方

式互不影响。在仓库前面开辟了新的道路去接入高速枢纽。这样就满足了两种不同的需求，如图10-8和图10-9所示。

图10-8 旁路传统访问

图10-9 两种方式访问的后端存储网络

10.1.3 万物归一——网络文件系统

微软老道在吸取瓜董的思想之后，终于创立了自己的理论。武当的仓库就像一个大的卷，一个大的磁盘阵列。它可以划分出多个LUN供多个使用者使用。而每个使用者必须有自己的文件系统，因为这个LUN只是一个卷设备，只提供了不计其数的房间存放货物。至于怎么存放货物。需要由使用者自己决定和管理，也就是用文件系统来管理卷，像理货员作货物记录一样。

微软老道把瓜董的思想用在了存储上，他把文件系统的功能从使用者处迁移到了磁盘阵列之上，让磁盘阵列自己管理存储空间。而对外提供统一的用户接口（货物存取单据v2.0），使得使用者不用再记录某某文件和卷上扇区或者簇块的对应关系，这个工作统统由盘阵上的集中式文件系统模块处理。使用者只需通过网络告诉这个文件系统需要存取什么文件，长度是多少就可以了。具体存取数据的过程，由集中式文件系统来做，使用者只需等待接收数据就可以了。同样，在存文件的时候，使用者只需告诉文件系统要存哪些数据，提供一些文件名、长度、哪个目录下等信息就可以了，至于文件存到卷的哪些空余扇区完全由盘阵上的文件系统逻辑来处理，使用者不必关心。

位于盘阵上的集中式文件系统得益于包交换网络，可以同时处理多个使用者的请求。它可以给每个使用者提供各自的文件夹目录，并且可以为这些目录限定允许存放的最大数据量。总之，文件系统可以实现的

任何功能都可以在盘阵上实现。

1. 网络文件系统

使用者如何与盘阵上的集中式文件系统进行交互呢？当然是通过网络来传递数据。由于一直以来的习惯，以太网加TCP/IP成为了首选的网络方式。除了底层传输网络，还必须定义上层的应用逻辑。

针对上层逻辑，微软定义了自己的一套规范，叫做CIFS（Common Internet File System），意思是Internet范围的FS。Linux和UNIX系统使用了另一种方式，称为NFS（Network File System），这些上层协议都是利用TCP/IP协议进行传输的。

以上描述的模型统称为“网络文件系统”。这种文件系统逻辑不是在本地运行，而是在网络上的其他节点运行，使用者通过外部网络将读写文件的信息传递给运行在远端的文件系统，也就是调用远程的文件系统模块，而不是在本地内存中调用文件系统的API来进行。所以网络文件系统又叫做远程调用式文件系统，也就是RPC FS（Remote Procedure Call File System）。

相对于SAN来说，这种网络文件系统不仅磁盘或卷在远程节点上，连文件系统功能也搬运到了远程节点上。本地文件系统可以直接通过主板上的导线访问内存来调用其功能。而网络文件系统只能通过网络适配器上连接的网线而不是主板上的导线来访问远端的文件系统功能。

网络文件系统在网络上传递的是些什么内容呢？下面用抓包的方式来分析一下。

2. CIFS协议网络包分析

在某个用CIFS访问方式的目录下新建一个文本文档，然后将它删除。此过程中抓取网络流量。在CIFS方式下，仅仅上述两个动作，就引发了网络上数百个包的流量（如图10-10所示），可见CIFS是一个开销非常大的NAS协议。这里就不一一分析每个包了。

图10-10 CIFS协议交互的数据包

3. NFS协议网络包分析

在一台Linux客户端上使用NFSv3来Mount一台NFS服务器上的某个目录到本地的/mnt目录下。进入这个目录，然后用touch a命令来创建一个名为a的文件，然后执行vi a，进入编辑模式，不作任何修改，退出，之后用rm a命令来删除这个文件。其间抓取网络上的流量。

提示： 10.128.132.45是NFS客户端的IP地址，10.128.132.175是NFS服务器端的IP地址。分析结果已经去除了不必要的TCP包以及TCP_ACK等包。

图10-11显示的是这个过程中网络上双方所交互的主要数据包。

图10-11 NFS方式下网络上的数据包

可以看到，NFS协议的开销远远小于CIFS协议。完成相似动作，NFS只需要交互十几个包即可。下面来分析每个包的作用。

Frame1（如图10-12所示）：客户端在创建文件之前，首先做了一次lookup操作，来查找当前目录中是否已经有同名文件；如果有，则拒绝创建。图中的DH表示Directory Handle，是一个32字节长的字段，这个值用来指代目录名称。在第一次访问某个目录时，NFS服务端会动态

分配这个值，将其通知给客户端。随后的访问请求中，客户端将使用这个值而不是目录名称来向NFS服务端发起针对这个目录的请求。

图10-12 客户端的Lookup请求

提示： 图中DH值的hash值为0x98f8d6bb，为了表示方便，抓包软件将其hash成一个4字节的值，这个hash值并不是存在于网络包中的。本例中，/mnt目录被指定的DH值的hash值就是0x98f8d6bb。

Frame2（如图10-13所示）：为NFS服务端对Frame1的回应。通过ERR_NOENT可以判断出当前目录并没有名为a的文件。

图10-13 Lookup请求的回应

Frame4（如图10-14所示）：客户端随即发起了“Create Call”，创建“a”文件。

图10-14 客户端的Create请求

Frame5（如图10-15所示）：NFS服务端对客户端Frame4的回应。创建成功，服务端返回File Handle（FH）的hash值为0xf03ce91c。FH与DH一样，在数据包中实际上也为一个32字节长的字段。为了表示方便，抓包软件将其hash成一个4字节的值。随后的交互中客户端不会用文件名来向服务端请求操作，而全部用这个File Handle来指代。

图10-15 Create请求的回应

Frame6（如图10-16所示）：文件“a”创建成功之后，出于保险起见，应用程序一般都会紧接着查询一下文件属性，顺便确认文件是否创

建成功。“GetAttr Call”就是用来查询文件属性用的一种RPC call。从图中可以看到对应的FH值为0xf03ce91c，NFS服务端收到这个值就会自动对应成文件“a”。

图10-16 客户端的GetAttr请求

Frame7（如图10-17所示）：NFS服务端对Frame6的回应。包中可以看到文件的umask访问权限以及atime、mtime、ctime属性。

图10-17 GetAttr请求的回应

Frame8（如图10-18所示）：紧接着NFS客户端发起了一个查询/mnt目录属性的请求。因为Handle的值为0x98f8d6bb，所以可以判断这个GetAttr Call是针对/mnt目录的。

图10-18 客户端针对/mnt目录的GetAttr请求

Frame9（如图10-19所示）：NFS服务端对Frame8的回应。

图10-19 针对GetAttr请求的回应

Frame10（如图10-20所示）：NFS客户端发起一个在/mnt目录中查找文件“a”的请求。这里由于是查找操作，客户端会假设不知道“a”文件的FH值，而只知道/mnt目录的DH值，所以文件名“a”使用的就是ASCII码的“a”。

图10-20 客户端的Lookup请求

Frame11（如图10-21所示）：NFS服务端根据Frame10中请求的回应找到这个文件，FH值是0xf03ce91c。

图10-21 针对Lookup请求的回应

Frame12（如图10-22所示）：NFS客户端发起一个SetAttr Call的请求，这个请求的目的是为了改变文件属性。可以看到客户端请求将文件的atime和mtime改为服务端当前的系统时间。

图10-22 客户端的SetAttr请求

Frame13（如图10-23所示）：对Frame12的回应。可以看到atime、ctime在之前和之后的不同，当然，时间差别都在微秒级，因为这一连串的请求其实是在很短的时间里发出去并得到应答的。

图10-23 针对SetAttr请求的回应

Frame15（如图10-24所示）：由于在客户端执行了“rm a”的命令，所以客户端发起一个对/mnt目录的访问请求。其实客户端在抓包期间一直处于/mnt目录中，至于为何要重新发起Access请求，与具体应用程序的代码有关，可能开发者为了确认/mnt目录的DH值没有过期，所以重新试探访问。

图10-24 客户端针对/mnt目录的Access请求

Frame16（如图10-25所示）：NFS服务端对Frame15的回应。

图10-25 针对Access请求的回应

Frame18（如图10-26所示）：此时，客户端由于输入了“rm a”命令，所以客户端首先要查询一下文件“a”的权限，因此客户端发起一个GetAttr的请求来查询文件“a”的权限。

图10-26 客户端的GetAttr请求

Frame19（如图10-27所示）：NFS服务端对Frame18的回应。

图10-27 针对GetAttr请求的回应

Frame20（如图10-28所示）：NFS客户端发起对文件“a”的Access请求。至于为何在删除文件之前要发起Access请求，与编码习惯有关。

图10-28 客户端的Access请求

Frame21（如图10-29所示）：NFS服务端对Frame20的回应。

图10-29 针对Access请求的回应

Frame23（如图10-30所示）：NFS客户端发起Remove请求。可以看到Remove请求中并没有使用FH，而是直接使用了文件名。

图10-30 客户端的Remove请求

Frame24（如图10-31所示）：NFS服务端对Frame23的回应。

图10-31 针对Remove请求的回应

可以看到，基于NAS的数据访问，客户端并不关心文件存放在磁盘的哪些扇区，这些逻辑全部由NAS服务端处理，客户端向NAS设备发送的只有各种文件操作请求以及实际的文件流式数据。

提示：大家可以在12.4节看到有关ISCSI的抓包分析，可以作一下对比，两者交互的语言完全不同。

10.1.4 美其名曰——NAS

这种带有集中式文件系统功能的盘阵叫做网络附加存储（Network Attached Storage, NAS）。

提示： NAS不一定是盘阵，一台普通的主机就可以做成NAS，只要它自己有磁盘和文件系统，而且对外提供访问其文件系统的接口（如NFS、CIFS等），它就是一台NAS。常用的Windows文件共享服务器就是利用CIFS作为调用接口协议的NAS设备。一般来说，NAS其实就是处于以太网上的一台利用NFS、CIFS等网络文件系统的文件共享服务器。至于将来会不会有FC网络上的文件提供者，也就是FC网络上的NAS，就要看是否有人尝试了。

1. SAN和NAS的区别

前面说过，SAN是一个网络上的磁盘，NAS是一个网络上的文件系统。

提示： 根据SAN的定义，即“存储区域网络”，SAN其实只是一个网络，但是这个网络内包含着各种各样的元素，如主机、适配器、网络交换机、磁盘阵列前端、盘阵后端、磁盘等。应该说，SAN是一个最大的范围，涵盖了一切后端存储相关的内容。所以从这个角度来看，SAN包含了NAS，因为NAS的意思是“网络附加存储”，它说的是一种网络存储方式，这样它就没有理由不属于SAN的范畴。所以，我认为SAN包含NAS。长时间以来，人们都用SAN来特指FC，特指远端的磁盘。那么，假设设计出一种基于FC网络的NAS，此时SAN代表什么呢？

会产生滑稽的矛盾。但是，似乎还真想不出一一种更简便、更直观的叫法来称呼“FC网络上的磁盘”这个事物。到此我也陷入定义的漩涡了，所以我们最好还是入乡随俗。本书之后的文字中，就把FC网络上的磁盘叫做SAN，把以太网上的文件系统称为NAS。这里就是提一下，不要被表象所迷惑。

2. FTP服务器为什么不属于NAS

我们必须明白什么是网络文件系统，网络文件系统与本地文件系统的唯一区别，就是传输方式从主板上的导线变成了以太网，其他方面包括调用的方式对于上层应用来说没有任何改变。

这就意味着，一旦用户挂载了一个网络文件系统目录到本地，那么他就可以像使用本地文件系统一样使用网络文件系统。

在Windows系统中，可以直接双击共享目录中的程序将其在本机运行（实际上是先通过以太网将这个程序文件传输到本地的缓存，然后才在本地执行，而不是在远端执行）。而FTP无法做到这一点，FTP不能实现诸如挂载等动作，它不是实时的。只有通过FTP将文件传输到本地的某个目录之后才能执行，而且这个程序执行需要的所有文件都必须在本机。

而网络文件系统则不然，即便某个本地执行的程序需要访问远端的某些文件，它也可以直接访问远端的文件，不需要预先将数据复制到本地再访问。所以，FTP、HTTP和TFTP等文件服务并不属于网络文件系统，也不属于NAS。

3. 普通台式机可以充当NAS吗

完全可以，只要具备NAS的特性，就可以充当NAS。

NAS必须具备的物理条件如下。

不管用什么方式，NAS必须可以访问卷或者物理磁盘。

NAS必须具有接入以太网的能力，也就是必须具备以太网卡。

普通台式机具备了这两个条件，就可以充当NAS。我们只要编写程序从磁盘提取或者存放数据，记录好这些数据的组织方法，然后通过网络文件系统协议规定的格式进行发送或接收，就可以实现NAS的功能。或者可以直接在操作系统上编程，直接利用操作系统已经实现好的文件系统和网络适配器驱动程序，所要做的只是利用操作系统提供的足够简单的API编写网络文件系统的高层协议逻辑即可。

10.2 龙争虎斗——NAS与SAN之争

10.2.1 SAN快还是NAS快

很多人都在问，到底是SAN快还是NAS快？要解答这个问题，方法非常简单，和百米赛跑一样，只要计算起点到终点的距离、耗时、开销就可以了。

SAN的路径图如图10-32所示。

图10-32 SAN方式路径图

NAS的路径图如图10-33所示。

图10-33 NAS方式路径图

显然，NAS架构的路径在虚拟目录层和文件系统层通信的时候，用以太网和TCP/IP协议代替了内存，这样做不但增加了大量的CPU指令周期（TCP/IP逻辑和以太网卡驱动程序），而且使用了低速传输介质（内存速度要比以太网快得多）。而SAN方式下，路径中比NAS方式多了一次FC访问过程，但是FC的逻辑大部分都由适配卡上的硬件完成，增加不了多少CPU开销，而且FC访问的速度比以太网高。所以我们很容易得出结论。如果后端磁盘没有瓶颈，那么除非NAS使用快于内存的网络方式与主机通信，否则其速度永远无法超越SAN架构。

目前万兆以太网已经逐渐普及，如果拿万兆和8GbFC相比，底层裸速率一定是万兆的要快，此时就要看上层的交互效率了。对于NAS访问协议，每一笔针对文件的操作，比如Create（）/Open（）、Lookup（）、Getattr（）、Setattr（）、Read（）/Write（）等，都需要

经过网络传输，而对于SAN来讲，由于文件系统运行在Host本机，大部分文件系统元数据也都可以被缓存到Host本地内存，所以大部分针对文件的非读写操作（比如Open（）/Lookup（）等），都可以在本地内存完成调用，此时速度非常快，但是这些操作在NAS环境下，就得通过网络远程向NAS Server发起请求，时延激增，虽然NAS Client端也有一小部分缓存，但是毕竟不如整个Host本地FS的缓存占据的容量大。当你的应用场景多是这类IO请求的话，那么性能远比不上SAN，但是你的场景如果是Read（）/Write（）请求居多，也就是读写实际数据的请求占多数的时候，此时速度就大多取决于底层物理链路的带宽了。

既然NAS一般情况下不比SAN快，为何要让NAS诞生呢？既然NAS不如SAN速度快，那么它为何要存在呢？具体原因如下。

NAS的成本比SAN低很多。前端只使用以太网接口即可，FC适配卡以及交换机的成本相对以太网卡和以太交换机来说是非常高的。

NAS可以解放主机服务器上的CPU和内存资源。因为文件系统的逻辑是要靠CPU的运算来完成的，同时文件系统还需要占用大量主机内存用作缓存。所以，NAS适合用于CPU密集的应用环境。

由于基于以太网的TCP/IP传输数据，所以NAS可扩展性很强。只要有IP的地方，NAS就可以提供服务，且容易部署和配置。

NAS设备一般都可以提供多种协议访问数据。网络文件系统只是其提供的一种接口而已，还有诸如HTTP、FTP等协议方式。而SAN只能使用SCSI协议访问。

NAS可以在一台盘阵上实现多台客户端的共享访问，包括同时访问某个目录或文件。而SAN方式下，除非所有的客户端都安装了专门的集

群管理系统或集群文件系统模块，否则不能将某个LUN共享，强制共享将会损毁数据。

经过特别优化的NAS系统，可以同时并发处理大量客户端的请求，提供比SAN方式更方便的访问方法。

多台主机可以同时挂接NFS上的目录，那么相当于减少了整个系统中文件系统的处理流程，由原来的多个并行处理转化成了NFS上的单一实例，简化了系统冗余度。

10.2.2 SAN好还是NAS好

关于IO密集和CPU密集说明如下。

- **CPU密集：**指的是某种应用极其耗费CPU资源，其程序内部逻辑复杂，而且对磁盘访问量不高。如超频爱好者常用的CPU测试工具就是这种应用，这种程序在运行的时候，根本不用读取磁盘上的数据，只是在程序载入的时候，读入一点点程序数据而已。进程运行之后便会使CPU的核心处于全速状态，这会造成其他进程在同一时间内只获得很少的执行时间，影响了其他程序的性能。在必要时，可以将多台机器组成集群来运行这种程序。
- **IO密集：**指的是某种应用程序的内部逻辑并不复杂，耗费的CPU资源不多，但是要随时存取硬盘上的数据。比如FTP服务器之类程序就是这种。
- **IO和CPU同时密集：**这种应用程序简直就是梦魇。为了获得高性能，大部分这类程序都不适合用单台机器运行，必须组成集群系统来运行这种应用程序，包括前端运算节点的集群和后端存储节点的集群。

显然，NAS对于大块连续IO密集的环境，要比SAN慢一大截，原因是积累效应。经过大量IO积累之后，总体差别就显现出来了。不过，如果要使用10Gb/s以太网这种高速网络，无疑要选用NAS，因为底层链路的速度毕竟是目前NAS的根本瓶颈。此外，如果是高并发随机小块IO环境或者共享访问文件的环境，NAS会表现出很强的相对性能。如果SAN主机上的文件系统碎片比较多，那么读写某个文件时便会产生随机小块IO，而NAS自身文件系统会有很多优化设计，碎片相对少。CPU密集的应用可以考虑使用NAS。

SAN与NAS有各自的优点和缺点，需要根据不同的环境和需求来综合考虑。

10.2.3 与SAN设备的通信过程

对于SAN方式来说，应用程序必须通过运行在服务器本机或者NAS设备上的文件系统与磁盘阵列对话。应用程序对本机文件系统（或NAS）说：“嗨，兄弟，帮我把/mnt/SAN目录下的SAN.txt文件传到我的缓冲区。”文件系统开始计算SAN.txt文件占用的磁盘扇区的LBA地址，计算好之后向SAN磁盘阵列说（用SCSI语言）：“嗨，哥们，把从LBA10000开始之后的128个扇区内容全部传送给我！”

SAN磁盘阵列接收到这个请求之后，便从它自身的众多磁盘中提取数据，通过FC网络传送给运行着文件系统程序的节点（服务器主机或NAS设备）。文件系统接收到扇区内容之后，根据文件系统记录截掉扇区多余的部分，将整理后的数据放入请求这个数据的应用程序的缓冲区。

10.2.4 与NAS设备的通信过程

应用程序通过操作系统的虚拟目录层直接与NAS设备对话：“嗨，兄弟，帮我把/mnt/NAS目录下的NAS.txt文件传过来。”或“嗨，兄弟，帮我把/mnt/NAS目录下的NAS.txt文件的前1024字节传递过来。”这些话被封装成TCP/IP数据包，通过以太网传递到NAS设备上。NAS接到这个请求之后，立即用自己的文件系统（NTFS、JFS2、EXT2、EXT3等）计算NAS.txt文件都占用了磁盘的哪些扇区，然后从自己的磁盘上用ATA语言或者SCSI语言向对应的磁盘存取数据（或自己安装FC适配卡，从SAN存储设备上存取数据）。

显然，NAS将文件系统逻辑搬出了主机服务器，成为了一个单独的文件系统逻辑运行者。如图10-34所示为从主机到NAS设备的IO全流程示意图，牵扯到了整个系统的IO路径以及路径上的每种元素和角色。

图10-34 网络文件系统IO全流程图

提示：关于系统IO路径的详细阐述请参考本书后面章节。

（1）首先，主机客户端通过NFS Client对NAS上的一个输出目录/nas/export进行Mount操作，将其Mount到了本地的/mnt/nas路径下。

（2）之后，主机客户端上某应用程序发起了对/mnt/nas/nas.txt文件的读取操作，读取从偏移量0字节开始往后的1024字节，也就是这个文件的前1024字节。这个动作是通过调用操作系统提供的文件操作API执行的，比如Read（）。

（3）这个IO请求被传送到了NFS Client处，NFS Client知道/mnt/nas路径对应的其实是NAS服务端的/nas/export这个输出目录，所以NFS Client将上层下发的这个读取请求封装成NFS协议规定的标准格式通过网络传送到NAS服务端。

(4) NAS服务端接收到这个读请求之后，将请求通过操作系统API传送给文件系统模块处理。

(5) 文件系统模块接收到针对/nas/export/nas.txt文件的读请求之后，首先查询缓存内是否有对应的数据，如果有，则直接返回结果；如果没有，则需要将这段字节所落入的底层存储空间的块信息取回，这个动作通过查询Inode表等元数据获得。当得到底层块地址之后，文件系统通过调用OS提供的API将对这些块的读请求发送给下游模块，也就是卷管理层。

(6) 卷管理层是将底层物理磁盘设备进行虚拟化封装的层次。当卷管理层收到针对某个卷某段LBA地址的请求之后，它要进行翻译，将目标虚拟卷的地址翻译为对应着底层物理磁盘块设备的地址；翻译完后，卷管理层将对应目标地址的请求再次通过调用OS API的方式发送给下游，也就是驱动程序层了。

(7) 块设备驱动是负责对相应块设备进行IO的角色，它将这些IO发送给SCSI CDB Generator，也就是SCSI指令的翻译中心。

(8) SCSI CDB Generator的职责是将对应的IO请求描述为SCSI协议的标准格式。之后，这些指令被发送到SCSI/FC适配器的驱动程序处。

(9) 设备驱动程序接收到这些SCSI指令之后，将其封装到对应的链路帧中通过内部总线网络或者外部包交换网络传送到目标。

(10) SCSI指令传送到目标设备，目标设备执行相应指令并返回结果。

10.2.5 文件提供者

NAS可以看作一个Filer。Filer这个词是著名NAS设备厂商NetApp对其NAS产品的通俗称呼。它专门处理文件系统逻辑及其下面各层的逻辑，从而解放了服务器主机。服务器主机上不必运行文件系统逻辑，甚至也不用运行磁盘卷逻辑，只需运行目录层逻辑（UNIX系统上VFS层、Windows系统上的盘符及目录）即可。把底层的模块全部交由一个独立的设备来完成，这样就节约了服务器主机的CPU资源和内存资源，从而可以专心地处理应用层逻辑了。

NAS网关就是这样一种思想。NAS网关其实就是一台运行文件系统逻辑和卷逻辑的设备，可以把它想象成一个泵，这个泵可以从后端接收一种格式（以LBA地址为语言的指令和数据格式），经过处理后从前端用另一种格式（以文件系统为语言的指令和数据格式）发送出去，或执行反向的过程。可以把这个泵接入任何符合条件的网络中，以实现它的功能。我们可以称它为文件系统泵，或者Filer。可以把SAN设备称为Diskar（专门处理磁盘卷逻辑），把服务器主机称为Applicationer（专门处理应用逻辑）。如果某个设备集成了Filer和Diskar的功能，并将其放入了一个机箱或者机柜，那么这个设备就是一个独立的NAS设备。如果某个设备仅仅实现了Filer，而Diskar是另外的独立设备，那么这个只实现了Filer的设备就称为NAS网关或NAS泵。

图10-35显示了这个泵接入网络之后发生的变化。

图10-35 NAS泵

然而，目前NAS的用途并不如SAN广泛，主要原因是NAS的前端接口几乎都是千兆以太网接口，而千兆以太网的速度也不过100Mb/s，除

去开销之后所剩无几。而SAN设备的前端接口目前普遍都是4Gb/s的速度，可以提供400Mb/s的带宽。FC现在已有8Gb/s速率的接口出现，而10Gb/s以太网也初露端倪。不久的将来，NAS必定会发起另一轮进攻。

10.2.6 NAS的本质

本地文件系统负责将硬盘上的数据包装、展现，以及提供调用接口。文件系统将数据展现为文件和目录的形式，调用接口包括读、写、删、创建等。

一般情况下，应用程序是在内存中调用这些接口从而让文件系统实现对应功能的。如Read（）、Write（）、API等。

如果应用程序不在本机运行，而是运行在网络的另一端，同时依然想保留原有的访问文件的方式。此时就需要允许远程机器上的应用程序可以把这些调用指令通过网络打包传送过来，这些指令必须双方都能够识别，为此，NFS和CIFS协议诞生了。二者统称NAS协议或者NAS。

多个远程的应用程序可以同时对本地文件系统发起IO请求。这样，本地这台服务器就变成了一台文件服务器了，NAS设备就这么诞生了。

所以NFS和CIFS又被称为“网络文件系统”，其实它们只是一种规定如何将文件操作指令及结果在双方之间传送和控制的协议。网络上只有协议，没有文件系统，文件系统都在本地。“通过外部网络而不是计算机内部总线来传递文件读写指令的系统”是对网络文件系统最准确和最本质的描述。

10.3 DAS、SAN和NAS

人们将最原始的存储架构称为DAS Direct (Dedicate) Attached Storage，直接连接存储，意思是指存储设备只用于与独立的一台主机服务器连接，其他主机不能使用这个存储设备。如PC中的磁盘或只有一个外部SCSI接口的JBOD都属于DAS架构。

纵观武当仓库的改革过程，恰恰正是一个从DAS（仅供自己使用）到SAN（出租仓库给其他租户使用），再到NAS（集中式理货服务外包）的过程。

到此，DAS、SAN、NAS形成了存储架构的三大阵营，且各有其适用条件，形成了三足鼎立之势！

提示：但是，大家一定要牢记，SAN (Storage Area Network) 是一种网络，而不是某种设备。只要是专门用来向服务器传输数据的网络都可以称为SAN。所以，NAS设备使用以太网向主机提供文件级别的数据访问，那么以太网就是SAN。由于在高端领域NAS的使用不如FC SAN设备多，所以人们习惯地称FC SAN架构为SAN。我们也顺从习惯，但是一定要明白其中的门道。

10.4 最终幻想——将文件系统语言承载于FC网络传输

既然SCSI语言及数据可以用FC协议传递，文件系统语言可以用以太网传递，那么文件系统语言能不能用FC传递呢？再者SCSI语言能不能通过以太网来传递呢？完全可以，而且后者已经被实现并被广泛应用了，这些内容将在第12章讲解。在4种组合中，只有将文件系统语言承载于FC网络传输这个想法没有被实现，或者说有人实现了但是没有听说。

现在来看前者。试想一下，如果用FC协议传递文件系统语言和数据而不是磁盘卷语言及其数据，那么开销会小很多。文件系统语言毕竟是比较上层和高级的语言，相对于底层磁盘卷语言来说，它非常简单。

读写某个文件，如果用高层语言只需要描述关于这个文件的信息即可。但是如果使用低级语言，可能要发送很多的语句。如果这个文件在磁盘上形成了很多碎片的话，发送的IO指令将不计其数。这就像C语言被编译器编译成汇编语句一样，文件系统语言是高级语言，比如“将C盘下C.txt文件传送过来”这一句高级语言，会被翻译成更低级的多条SCSI语言。利用比内存总线速度慢得多的FC网络来传送这些低级语言，无疑是非常浪费资源的。低级语言就应该在内存中传递而不是外部低速网络上，这样才能达到性能最大化。

目前普遍的架构是文件系统和磁盘控制器驱动程序都运行在应用服务器主机上。文件系统向卷发送的请求是通过内存来传递的，而主机向磁盘（LUN）发送的请求是通过FC网络来传递的，后者速度显然比前者慢很多。如果将“文件系统、磁盘控制器和磁盘（SAN盘阵或者本地

磁盘)”这三个部件，整体搬到应用服务器主机外部，成为一个独立的NAS系统。然后这个NAS设备用FC协议与服务器主机通信，FC协议上承载的是文件系统语言，这就既保证降低了服务器的开销，又保证了数据传输速度。

也许受限于技术或者商业和市场的限制，File system（或File）over FC（FC网络文件系统）始终没有被提出或开发出来。不过在10G以太网普及之后，相信FC设备也会因为高成本、高专用化、兼容性问题逐渐被淘汰。目前市场上的NAS设备普遍采用以太网加上TCP/IP的模式来传送文件系统指令，我们可以称这种架构为File system over Ethernet（FSoE）。

提示：关于协议之间相互杂交的论题，在后面会详细讨论。

10.5 长路漫漫——存储系统架构演化过程

下面总结一下系统架构演化过程中的10个阶段。

10.5.1 第一阶段：全整合阶段

图10-36是一个最原始的普通服务器的架构，所有部件和模块都在一个服务器机箱中。这属于DAS架构，因为主机机箱内的磁盘只被本机使用。

图10-36 全整合阶段

10.5.2 第二阶段：磁盘外置阶段

图10-37所示的架构，是将磁盘置于服务器机箱外部的情况。这种架构依然属于DAS架构，因为存储系统只被一台主机使用。

图10-37 磁盘外置阶段

10.5.3 第三阶段：外部独立磁盘阵列阶段

图10-38是服务器主机通过普通SCSI线缆连接外部独立磁盘阵列的情况。这种简单的SCSI接口盘阵只能供一台或者几台（如果盘阵提供多个外部SCSI接口的话）主机接入，称不上彻底的网络化，但可以被称为SAN，因为这种架构已经开始显现网络化萌芽了。

图10-38 外部独立磁盘阵列阶段

10.5.4 第四阶段：网络化独立磁盘阵列阶段

图10-39是一台服务器用FC网络连接FC接口磁盘阵列的情况。图中磁盘阵列真正成为包交换网络上的一个节点，可同时被多个其他节点访问，是向彻底网络化进化的里程碑。这种架构是彻彻底底的SAN架构。

图10-39 网络化独立磁盘阵列阶段

10.5.5 第五阶段：瘦服务器主机、独立NAS阶段

图10-40中，服务器主机用以太网与NAS设备进行通信从而存储数据。在瘦服务器阶段，文件系统之下的所有层次模块位移到了外部独立设备中。主机得到了彻底的解放，专门处理业务逻辑，而不必花费太多资源去处理底层系统逻辑。

图10-40 瘦服务器主机、独立NAS阶段

10.5.6 第六阶段：全分离式阶段

在NAS设备的后端可以用包含在自己机箱内的硬盘，也可以用并行SCSI来连接磁盘，还可以用FC协议来连接SAN盘阵来获得LUN（NAS网关）。在这个阶段中，所有部件彻底地分离了，每个部件都各司其职，中间通过不同的网络方式通信，前端用以太网，后端用FC网，如图10-41所示。

图10-41 全分离式阶段

10.5.7 第七阶段：统一整合阶段

在图10-42的环境中，既有纯SAN的磁盘阵列，来利用自己机箱或扩展柜内的磁盘，又有NAS网关设备，不但可以利用本地磁盘，还可以

向SAN磁盘阵列“租赁”若干LUN卷。另外还有一台多协议磁盘阵列，它既可以向外提供FC协议的连接方式（承载磁盘卷语言），又可以提供以太网的连接方式（承载文件系统语言），这种设备是SAN与NAS融合的结果。服务器可以选择直接用磁盘卷语言访问SAN磁盘阵列上的LUN，用运行在服务器上的文件系统程序管理磁盘卷，也可以选择直接通过以太网络访问NAS设备上的目录，用文件系统语言向NAS发送指令。

这个架构最终就是一个整合的架构，既有纯SAN和纯NAS，又有被整合到一起的SAN和NAS统一存储设备。

第七阶段也是目前IT系统所应用的架构。

图10-42 统一整合阶段

10.5.8 第八阶段：迅速膨胀阶段

也就是Scale-Out架构的x86集群阶段，使用大量的节点来组成一个对外统一的视图。从物理可见的角度上讲，是大规模并行集群；从较高层的角度来讲，由单个的、无意识的节点来组成一个庞大的整体有意识的整体，这就变成了云的一种特征了。不管是SAN还是NAS，它们此时被承载到同质化的底层集群上，只是前端输出的访问协议不同而已。

此时整个系统中的节点数量规模将会空前庞大，系统内部的能量将迅速爆发，如图10-43所示。

图10-43 迅速膨胀阶段

10.5.9 第九阶段：收缩阶段

当大量的节点迅速膨胀之后，能量最终会被消耗殆尽，随着新技术的不断产生，当机械硬盘被芯片所取代，并且大规模集成电路技术不断地发展，操作系统中的所有底层模块均用专用硬件电路实现。CPU所做的仅仅是进行业务逻辑计算。外部接口已经过渡到了全部使用无线网络进行通信，包括连接显示器（或许已经被3D眼镜所取代）。

目前，Intel公司已经在其CPU上集成了显示芯片，使计算机不再需要加装独立的显卡，也不用占用主板上的空间来集成显示芯片。Sun公司将多个万兆以太网芯片集成在了最新的CPU上。Bluearc公司的存储产品更是已经将整个操作系统的所有逻辑都做到了FPGA芯片中。这些新技术正在发芽，随着节点数量的膨胀，其底层却在向着收缩的方向演进。IT架构正处于缓慢收缩之中。图10-44所示为收缩阶段系统架构示意图。

图10-44 收缩阶段

10.5.10 第十阶段：强烈坍缩阶段

长过了头就要收缩，阳气消散之后，阴实就会逐渐显露并占上风，整个系统会发生强烈坍缩。对于系统架构，坍缩之后的IT系统会节约大量空间和电力能量。随着集成电路技术发展一定程度之后，甚至可以将整个系统集成于一个微小的芯片中，即所谓的System on Chip（SOC）。IT系统将坍缩到这个芯片中，如图10-45所示。

图10-45 强烈坍缩阶段

纵观这10个阶段，从全整合阶段（占用很大空间和资源）到坍缩阶段（极小的空间极高的质量），这正是宇宙发展演化的规律。在一个极大的空间中包含了众多物质，然后通过不断演化，最终坍缩到一个小空

间内，同样包含了极大的质量。

到此，SAN已经不仅仅限于通过FC网络传递SCSI指令的架构了。SAN的概念，既然是“存储区域网络”的意思，那么就应该泛指参与主机服务器后端存储系统的所有部件，甚至包括NAS，我们可以将NAS看作SAN的一个分支架构。

SAN在IT系统架构进化到第十阶段之前还会继续存在。在第十阶段，SAN的概念或许会被赋予另外的意义。

10.6 泰山北斗——NetApp的NAS产品

NetApp公司掌握了全球最先进的NAS方面的相关技术，它的FAS系列产品统治了NAS市场的大部江山。FAS系列中的所有产品均运行Data ONTAP操作系统，这是NetApp专门开发的针对NAS的操作系统。

既然是NAS，其内部的文件系统层肯定是一个功能强大而稳定的层次。ONTAP系统中的文件系统名为WAFL，这是一个充满个性的文件系统。

NetApp自称其存储产品为“Filer”，下面就来看看Filer的四把杀手锏。

10.6.1 WAFL配合RAID 4

Write Anywhere Filesystem Layout (WAFL) 是NetApp公司开发的一种文件系统。这个文件系统最大的特点，也是其他所有文件系统都没有实现的特点，就是它能按照RAID 4的喜好来向RAID 4卷写数据。RAID 4由于其独立校验盘的设计，导致它只能接受顺序的写入IO而不能并发，所以它对于其他厂家的盘阵来说完全就是一个灾星，没有人敢用，也没有人愿意用。然而NetApp偏偏采用了它，而且通过WAFL的调教，RAID 4在FAS产品上发挥出了很好的性能。WAFL是怎么调教RAID 4的呢？

与其说是WAFL调教RAID 4，不如说是RAID 4逼迫WAFL就范。

RAID 4再不好也有可取的地方，如果IO写入有很大几率是整条写的形式，那么RAID 4便会表现得像RAID 0一样良好。不仅仅是RAID

4, 任何校验型的RAID, 如RAID 5、RAID 3, 只要是整条写, 便会产生极高的性能。

提示: 关于整条写的概念, 可参考本书第4章的内容。

既然校验型RAID最喜欢被整条写, 那么就不妨满足这个要求。WAFL就是这么被设计出来的。

图10-46所示的是一个5块盘组成的RAID 4系统。条带大小20KB, 条带在每个数据盘上分割出的segment大小为4KB。假如某一时刻应用要求WAFL写入三个文件, 如/tmp/file1、/tmp/file2和/tmp/file3, 其中file1大小为4KB, file2大小为8KB, file3大小为4KB。则WAFL便会计算出: 如果将这三个文件对应的数据写到一整条条带上, 就构成了整条写, 性能得到了提升。

图10-46 5块盘组成的RAID系统

所以, WAFL先在其元数据中做好记录: /tmp/file1对应数据块为A1, /tmp/file2对应数据块为A2+A3, /tmp/file3对应数据块为A4。然后将这三个文件的数据合并写入RAID。当然, 如果要求增加高并发度, 那么WAFL也可以将同一个文件对应的一定长度的块写入同一个硬盘。

实际上, 文件系统就应该适配底层的特性, 只有这样才能获得最优的性能。普通文件系统中, 在选取空闲块写入数据的时候, 并没有针对底层的RAID级别来做对应的优化。而WAFL中有一个专门的Write Allocation模块来负责优化。WAFL的做法无疑对文件系统的优化起到了领头和示范作用。

以上只是简略说明WAFL的思想, 实际操作中还需要考虑诸如元数

据的写入、块的偏移等很多复杂情况。

10.6.2 Data ONTAP利用了数据库管理系统的设计

我们知道，数据库管理系统是这样记录日志的：在某时刻，数据库管理系统接收到应用程序的SQL更新语句及其对应的数据，在将这些数据更新到缓存中覆盖原有数据的同时，将这个操作的动作以及对应的数据，以日志的形式记录到位于内存的日志缓冲区内。每当应用程序发起提交指令或每隔几秒钟的时间，缓冲区内的日志就会被写入到磁盘上的日志文件中，以防止意外掉电后造成的数据不一致性，同时将缓存中更新过的数据块写入磁盘。只有当日志被确实地写入到硬盘上的日志文件中后，数据库管理程序才会对上层应用返回执行成功的信号。

数据库系统是一个非常复杂的系统，也是一个对数据一致性要求极高的系统，所以数据库利用记录操作日志的方式来保证数据一致性的做法是目前普遍使用的，而且是实际效果最好的方法。NetApp的Data ONTAP操作系统，就是利用了数据库这种设计思想，它把向文件系统或卷的一切写入请求作为操作日志记录到NVRAM中保存，每当日志被保存到了NVRAM中，就会向上层应用返回写入成功信号。

为何要用NVRAM而不是文件来保存日志呢？

10.6.3 利用NVRAM来记录操作日志

数据库系统完全可以直接将操作日志写入磁盘，而不必先写入内存中的日志缓冲区，再在触发条件下将日志写入磁盘上的文件。然而这么做会严重降低性能，因为日志的写入是非常频繁的，且必须为同步写入。如果每条日志记录都写入磁盘，则由于磁盘相对于内存来说是慢速

IO设备，所以会造成严重的IO瓶颈。所以必须使用内存中的一小块来作为日志缓存。

但是一旦系统发生意外掉电，则内存中的日志还没来得及保存到硬盘就会丢失。在数据库再次启动之后，会提取硬盘上已经保存的日志文件中的条目来重放这些操作，对于没有提交的操作，进行回滚。这样，就保证了数据的一致性。

如果某个应用程序频繁地进行提交操作，则日志缓冲区的日志便会被频繁写入磁盘。在这种情况下，日志缓存就起不到多少作用了。幸好，对于很多需要访问数据库的应用程序来说，上层的每个业务操作一般都算作一个交易，在交易尚未完成之前，程序是不会发送提交指令给数据库系统的，所以频繁提交发生的频率不高。

然而，上层应用向文件系统中写数据的话，每个请求都是一次完整的交易。如果对每个请求都将其对应的操作日志写入磁盘，开销就会很大。所以NetApp索性利用带电池保护的RAM内存用作记录操作日志的存储空间。这样一来，不但日志写入的速度很快，而且不用担心意外掉电。有了这种电池保护的RAM（称为NVRAM似乎不合适，因为NVRAM不用电池就可以在不供电的情况下保存数据，而NetApp使用的是带电池保护的RAM，下文姑且称其为NVRAM）来记录操作日志，只要日志被成功存入了这个RAM，就可以立即通知上层写入成功。

一定要搞清楚日志和数据缓存的区别。日志只是记录一种操作动作以及数据内容，而不是实际数据块，前者比后者要小很多。实际数据块保存在RAM中而不是NVRAM中。

由于用RAM来保存日志，所以速度超级快，可以一次接收上千条

写入请求而直接向上层应用返回成功信号，待RAM半满或每10秒钟的时候，这些数据由WAFL一次性批量连续写入硬盘，保证了高效率。这也是NetApp的NAS为什么相对比较快的一个原因。

10.6.4 WAFL从不覆写数据

每当NVRAM中的日志占用了整个NVRAM空间的一半或每10秒钟也可能是其他的某些条件达到临界值的时候，WAFL便会将所有缓存在内存缓冲区内的已经改写的的数据以及元数据批量写入硬盘，同时清空操作日志，腾出空间给接下来的请求使用。这个动作叫做Check Point。

WAFL并不会覆盖掉对应区块中原来的数据，而是寻找磁盘上的空闲块来存放被更改的块。也就是说，所有由WAFL写入的数据都会写入空闲块，而不是覆盖旧块。另外，在Check Point没有发生的时候，或者数据没有全部被Flush之前，WAFL从来不会写入任何元数据到磁盘。

有了以上两个机制就可以保证在Check Point没发生之前，磁盘上的元数据所对应的实际数据仍然为上一个Check Point时候的状态。如果此时发生突然断电等故障，虽然可能有新数据已经被写入空闲块，但是元数据并未写入，所以磁盘上保存的元数据还是指向旧块（这也是为何旧块从来不会被WAFL覆盖的原因），数据就像没有变化一样，根本不用执行文件系统检查这种耗时费力的工作。一旦Check Point被触发，则WAFL先将缓存中的所有数据写入磁盘空闲块，最后才将元数据写入硬盘。一旦新的元数据写入了硬盘，则新元数据的指针均指到了方才被写入的新数据块，对应的旧数据块则变为空闲块（虽然块中仍有数据，但是已经没有任何指针指向它）。

这个特性使得NetApp的快照技术水到渠成，且性能良好。

10.7 初露锋芒——BlueArc公司的NAS产品

上文中所述的IT系统架构发展的第九阶段（收缩阶段），其代表就是软件的全芯片化。这些芯片不同于CPU，而是完全应用逻辑芯片，比如ASIC或更高成本的FPGA和 CPLD。这些芯片对于专用逻辑的运算速度远高于CPU，因为CPU是读取外部程序代码指令流来执行并生成结果的，而专用芯片则是通过读入原始数据信号，在经过内部逻辑电路之后直接生成了输出信号。一片频率100MHz的FPGA在运行专用逻辑的时候速度也会高于频率几GHz的CPU。图10-47所示为CPU的处理流程，而图10-48所示的是FPGA的处理流程。

图10-47 CPU的处理流程

图10-48 FPGA的处理流程

FPGA芯片的频率目前已经可以超越1GHz，其内部电路已经达到1000万门。其可容纳的逻辑更多、更复杂，处理速度越来越快。

目前有些厂家正在尝试利用FPGA来取代CPU。由于FPGA可重构计算的特性，人们认识到许多能发挥其特长的应用。比如在玩《Crysis》电脑游戏时，可将FPGA配置成128位的高性能3D图像处理器；当需要听高保真环绕立体声时，可将FPGA配置成专用的DSP处理器；在高层网络交换机需要支持新协议时，只需重新配置FPGA而不必更改任何硬件；在数字电视变更解码协议时，只要通过网络下载数据来重新配置FPGA；当从GSM网转到CDMA网时，也只需重新配置FPGA而不必更换手机了。

同样，存储产品公司BlueArc在其NAS产品Titan系列中，将其上运行的所有软件逻辑都写入了FPGA中。其产品将存储系统路径上的多个

模块也分别做成了可插拔式模块，包括前端网络接口模块、文件系统模块和后端网络接口模块。前端接口是面对客户端的接口；文件系统则是整个系统的处理中枢；后端接口则是连接磁盘扩展柜的接口。每个模块上均有多个FPGA芯片来处理各自的逻辑。图10-49显示了Titan各个模块之间的生态架构图。

图10-49 Titan存储产品内部架构示意图

图10-50为Titan产品实物图。后面有4个插槽模块，中间两个为文件系统模块，最上面和最下面的模块分别为前端网络模块和后端网络模块。

图10-50 Titan 存储产品实物图

且不说Titan这种全硬件架构是否成熟，其内部软件是否兼容性良好，抛开这些因素不谈，这种架构其实反映的是一种趋势，一种精神。

10.8 宝刀未老——Infotrend公司NAS产品

在中端SAN存储方面，似乎到处都可见到Infotrend公司的产品，因其优良的做工和中规中矩的软件特性，非常适合被OEM。如今Infotrend这员老将也推出了自己的双控统一存储产品——EonNAS，最大支持到256盘。同时支持用iSCSI、FC、NFS/CIFS方式访问，最大支持4个万兆以太网口，如图10-51所示。这款NAS系统采用的是双控可热插拔模块化设计，设计上还是很用心的。

图10-51 EonNAS3000控制器机头

对于NAS来讲，软件核心和功能胜过硬件规格。EonNAS底层采用ZFS作为存储软件核心。ZFS拥有多项特色，比如Raid-Z、重定向写、原生快照/克隆、远程复制、压缩、重删、数据一致性支持等等。这也是近年来ZFS几乎被所有二三线厂商用作存储底层的原因，因为它几乎搞定了SAN和NAS存储系统底层所需要做的多数工作。如图10-52所示为EonNAS3000配置界面。

图10-52 EonNAS3000配置界面

这里要着重介绍的一点是，EonNAS支持无缝Failover。多数双控NAS产品是不支持拷贝文件的同时发生控制器切换而同时又能保证文件拷贝过程继续的。很欣慰地看到EonNAS很好地做了支持，支持这个技术，需要双控之间同步已经打开的文件句柄等信息，还是比较有挑战性的。如图10-53和图10-54所示，当在执行文件拷贝动作时，强行Failover一台控制器，但是文件拷贝只是提示了重试或者跳过，并没有完全中断，此时点击重试即可恢复之前的拷贝状态。

图10-53 强行Failover

图10-54 拷贝可以继续执行

第11章 大师之作——大话以太网和**TCP/IP**协议

- 以太网

- **IP**

- **TCP**

- **UDP**

- **OSI**

在老祖宗的OSI宝典的基础上，后人不知哪位大侠，创立了自己的功夫秘笈，后广泛流传于江湖。这套秘笈，就是大名鼎鼎的“以太网和TCP/IP大法”。

11.1 共享总线式以太网

如何让多台PC之间能够相互连接并且相互发送信息呢？既然说到独立系统之间的互联，那就可以参考OSI来做。在参考OSI之前先参考三元素，即连起来、找目标、发数据。

11.1.1 连起来

“连”这个动作，包括了OSI的物理层和数据链路层。首先要找一种连接方式将所有节点连接起来。连接多个节点最简单的办法就是总线技术。总线就是一个公共的媒介。要交流，就必须提供交流所需的场所，这个场所就是总线。将总线想象成一根铜导线，每个节点都连接到这条线上。这样每个节点的信号，在总线上所有的其他节点都会感知到，因为良导体上的电位处处相等。

基于这个总线模型，早期的以太网是使用集线器（HUB）将每台PC都连接到它的一个接口上，所有这些接口通过集线器内的中继电路连接在一起。为什么需要中继器，为什么不可以直接物理连接在一起呢？使用中继器的主要原因有两个：第一，信号在总线上传输时受到干扰可能会迅速衰减。加了中继器后集线器将从一个接口收到的bit流复制到每个接口，这样就避免了信号衰减；第二，中继器可以防止由于不可知的原因，造成两个节点同时向总线上放置信号所造成的短路。图11-1是一个HUB模型。

图11-1 HUB模型

在数据链路层，以太网使用帧的形式来发送数据流。上层的数据流被封装成一个个的以太网帧，在总线上传播。

11.1.2 找目标

为了区分总线上的每个节点，每个节点都必须具有一个唯一的身份标志。以太网中，这个标志被称作Media Access Control (MAC) 地址，介质访问地址，即只有数据帧中包含这个地址，总线介质上拥有这个地址的接收方才知道这个数据帧是给自己的，从而才会将其保存到缓冲区内。实际上，每个以太网帧中都包含源MAC地址和目的MAC地址。

MAC地址是一个6字节（48b）长的字段，每个节点的网卡都有一个全球唯一的MAC地址，这个地址在网卡出厂时被固化在芯片中。

以太网就是利用MAC地址来区分每个节点的。

11.1.3 发数据

既然是总线方式联网，那么每个节点发出的信号，总线上的所有节点都会感知到，并且，同一时刻只能由一个节点的信号在总线上传递，如果同时有多个节点都向总线上传递信号，则各路信号之间就会发生冲突。比如，节点A在时间点T1时刻向总线上放置高电位信号（假设电位为0.5V），而节点B在T1时刻向总线上放置低电位信号（假设电位为0V），这样，就形成了电流通路，一方面造成短路，缩短设备寿命；另一方面总线上的其他节点所感知到的电位总是0。所以，任何情况下，都不能让多个节点同时向总线上放置信号。

有如下两种措施可以防止这种情况的发生。

- 集线器中的中继电路，会防止由于恶意破坏或者其他不可知的程序bug所导致的信号冲突。

- 在协议角度，从根本上杜绝这种情况的发生。

在总线上，每个节点利用载波侦听机制（CSMA）来检测当前总线上是否有其他节点的信号正在传播，一旦检测到信号，则暂时不发送缓冲区内的数据帧，并不断地侦听电路上的信号，一旦发现总线空闲，则立即向总线上放置信号，声明要使用总线，如果在完全相同的时刻，另一个节点也同样放置了信号，则两路信号会发生冲突，两个节点检测到冲突后，会撤销声明，继续回到侦听状态，这个过程叫做冲突检测（CD）。

但两个节点在同一时刻同时发出信号的几率很小，即使本轮声明失败，在下一轮争抢声明中，某个节点胜利的几率是很大的，而且以太网中的所有节点的优先权都是一样的，或者说以太网内没有优先权的概念，包括网关设备在内。而SCSI总线的优先级最高，因为SCSI协议本身就是一个Poll- Response型的协议，SCSI控制器要顺序寻找总线上的除自身之外的所有其他节点，它的优先级最高。所以对于以太网来说，完全不必担心某个节点永远也抢不到总线使用权（俗称“总线饿死”）的情况。

另外，也不必担心这种CSMA/CD机制会耗费过多的时间，由于这种机制不是靠CPU执行代码来实现的，而完全是靠电路逻辑执行的，所以速度都在微秒级，宏观上不会感觉到延迟。

如果一条总线上的节点过多，则发生冲突的几率就越大，造成的开销和延迟也越大。另一方面节点越多，每个节点使用总线的平均时间也会降低，宏观上，也就造成了每节点的可用带宽降低。所以共享总线方式的网络，可接入的节点数量是有限的。

节点取得了总线使用权之后，便开始发送数据帧，也就是将数据帧的比特流转换成电信号，以一定的间隔速度放置到总线上，对于10Mb/s以太网来说，每隔一千万分之一秒，总线上的信号就被放置一次，直到整个数据帧被传播完毕，总线空闲，然后节点再发起新一轮的CSMA/CD过程。

在一个节点向总线上传播数据的同时，所有其他节点都会将总线上正在传播的信号保存到各自的缓冲区内，形成一帧一帧的数据。也就是说，共享式以太网中，任何一个节点所发送的数据，其他所有节点都会收到。

这岂不是没有隐私可言了么？

的确，只要在HUB上的任何一个节点上安装一个抓包软件，就可以抓取总线上的信号，看到任何源节点发送到任何目的节点的数据。

比如，节点A想与节点B通信，节点B怎么知道当前总线上的数据是给它的呢？我们上面讲过，每个节点都有一个MAC地址，节点A若想将数据传送给节点B，就会在数据帧中的特定字段填入节点B的MAC地址（目的MAC地址），并且在特定字段填入节点A自己的MAC地址（源MAC地址），这个数据包被所有人收到，当然包括B节点。B节点检测这个帧的目的MAC地址，与自身的MAC地址作对比，如果相配，则得知这个帧是给自己的，并且通过检测源MAC地址段，B会知道这个帧是节点A给的。而节点C此时也收到了这个帧，它也会对比自己的MAC与帧中的目的MAC字段，发现不匹配，所以知道这个帧不是给自己的，立即将其丢弃。

以太网定义了一种特殊的MAC地址，这个地址的所有48位的值都

为1。这个地址对应了总线上的所有节点，也就是说，如果某个数据帧中的目的地址是这个特殊的MAC地址，则所有接收到这个帧的节点都会保存起来并提交上层协议处理，而被这个地址称为广播地址。

11.2 网桥式以太网

前面描述了共享总线型以太网。会发现，一个人说话所有人都听到，这不安全，也没必要浪费资源。必须发明一个机制来改进，网桥的出现，初步降低了这个问题所带来的影响。

网桥也有多个接口，外观甚至和HUB一样，但是里面的电路加入了逻辑运算电路和智能化的东西，不像HUB一样，所有接口复制总线上的数据。

网桥把它上面的接口分了很多组，每一个组中的接口都在一条独立的总线上，不同的组使用不同的总线。也就是说，除非网桥转发过去，否则某个组中总线上的数据，不会被另外的组收到。这样就减小了冲突域，相当于本来10个人的小组，分成了多个子讨论组，一个讨论组在讨论的同时，另一个讨论组也可以讨论，两个组完全物理上隔离。这样，接入相同数目的节点，但是性能却比HUB好多了。图11-2为一个网桥模型。

图11-2 网桥模型

如图11-2所示，节点A与B在总线1（冲突域1）中，总线1与总线2是隔离的，如果A与B通信，则只会在总线1上节点，A与B通信的同时，总线2上的C与D也可以通信。这样，每个冲突域中所包含的节点数目降低了，所以相对HUB来说提高了性能。

但是如果A想与D通信，怎么办呢？毫无疑问，桥接芯片必须将A发出的数据帧从总线1复制到总线2上；反之亦然。那么，桥接芯片如何知道某个数据帧是A发送给D的，而不是B的呢（A发送给B的数据帧不能

被复制到总线2，否则就和HUB一样了）？显然，桥接芯片必须对数据帧进行分析，分析其中的目的MAC地址，如果某个数据帧的源节点在总线1上，其目的MAC地址对应的节点也在总线1上，那么芯片就不会复制这个数据帧到其他总线，但是如果目的MAC对应的节点与源节点不在一个总线上，则桥接芯片会将这个数据帧转发到目的节点所在的总线，从而让目的节点收到。

为了做到这一点，网桥必须知道某条总线上到底包含了哪些MAC地址，这样才能做到根据目的MAC与总线的对应关系来决定某个数据帧是否需要转发，转发到哪条总线。桥接芯片是通过动态学习来填充“MAC—总线”对应表的。

每当总线上有数据帧通过，桥接芯片就去提取它的源MAC地址，放入对应表中（已经存在的记录将会忽略）。比如，当网桥刚刚加电的时候，这张对应表是空的。这时总线1上的A向B发送了一个数据帧，因为是共享总线，所以桥接芯片也会同样收到这个帧，网桥立即提取这个帧中的源MAC地址，假如这个MAC值为：00-40-D0-A0-A3-7D，就向表中填入一条“00-40-D0-A0-A3-7D bus1”，这表示bus1总线上，有一个MAC地址为00-40-D0-A0-A3-7D的节点连接着。同样，所有总线上所收到的源MAC地址都会被记录到表中。如果从总线1上收到一个数据帧，经过查表发现其目的MAC在另一条总线上，则网桥将这个帧转发到那条总线。如果某个帧的目的MAC还没有被网桥学习到，也就是说网桥不知道这个MAC在哪个总线，则网桥便向除发送这个帧的总线之外的其他总线转发这个帧，直到学习到这个MAC条目。

强调： 网桥并不记录到底是哪个端口所连接的节点具有这个MAC，因为根本没有必要记录端口信息，数据帧只要进入这条总线，这条总线上所有端口下连接的节点就都会收到数据，

只记录“总线-MAC”统计表即可，所以没有必要记录“端口-MAC”表。

11.3 交换式以太网

HUB是一锅粥，网桥也好不到哪里去，只不过把这一锅大粥分成了多锅小粥而已。

随着硬件技术的提高，交换式以太网出现了。交换式以太网利用交换机替代了网桥。其实交换机也是一种网桥，一种特殊的网桥。普通网桥中，每个端口组（冲突域）中有不止一个端口，而交换机中，每个端口组中只有一个端口，也可以这么说，交换机上的每个端口都独占一条总线。这样交换机彻底隔离了冲突域，而不是仅仅减小了冲突域，也就是说，交换机的每个端口下都是一个独立的冲突域。

此外，交换机内部芯片，也不再被称作桥接芯片，而改成交换芯片。网桥芯片学习并记录“MAC-总线”对应表，那么顺藤摸瓜，交换机既然每个端口独占一个总线，理所当然的，交换芯片学习并记录的就是“MAC-端口”对应表。

交换机是以太网的最终实现形式。端口之间不再是共享总线通信，而是可以在任意时刻、任意两个端口之间同时收发数据。

HUB是没有大脑的，只有一个中继器。网桥进化出了大脑，但是智商不高。而交换机则拥有高智商的大脑和极快的运算速度。图11-3便是一个交换式模型。

图11-3 交换式以太网模型

11.4 TCP/IP协议

以太网的出现，给系统间互联提供了方便的方式。每个节点安装一块以太网适配器，上层程序只要将要发送的数据以及数据要到达的目的MAC地址告诉以太网卡，数据就可以通过以太网传输到目的，完成通信过程。

但是实际中，以太网并不是直接被应用程序使用来收发数据的。究其原因，就是因为以太网是一个没有传输保障机制的网络。首先以太网设备不会对数据帧进行校验纠错等措施。其次以太网设备一旦由于数据交换量过大，就可能造成缓冲区队列充满而主动丢弃数据帧，而且丢弃数据帧的同时，不会有任何措施来通知发送方。所以，以太网是一个不可靠的网络。应用程序如果直接调用以太网来发送数据，则必须忍受丢失数据的风险。

为了解决以太网的这个弱点，人们在以太网之上，增加了一个层次，在这个层次上，人们实现了以太网所不具有的功能。运行在这个层次上的程序，用了很多复杂的机制来保证发送给以太网适配器的数据包能够成功地到达对方，如果中途发生丢包现象，则本地会通过超时未得到确认等机制来重新发送该数据包。而应用程序由原来的直接调用以太网接口，改为调用这种位于以太网上层的新程序接口从而获得可靠的数据收发。

运行在这个新层次上的协议有很多，如NetBEUI、NetBIOS、IPX、TCP/IP等。这些协议，其下层调用以太网提供的服务，上层则向外提供新的调用接口，向应用程序提供可靠的网络传输服务。

在这些协议中，TCP/IP以其广谱的适用性、良好的性能以及能够良好地在超大规模网络上运行等优点，迅速地得到了普及，成为Internet网络所使用的通信协议。

11.4.1 TCP/IP协议中的IP

到MAC这一层，以太网已经实现了OSI的下三层，即物理层、链路层和网络层。以太网也只跨越了这3层，从第四层到第七层，以太网没有涉足（现在普遍认为以太网只作用到链路层，这是一个错误观点。造成这种误解的原因，是以太网一旦与TCP/IP结合，便沦落为TCP/IP协议的链路层，其第三层地址被以太网的IP所映射掉了，所以掩盖了其第三层的元素）。

由于以太网的天然弱点，使它不得不选择与TCP/IP协议合作，求助TCP/IP协议向上层应用程序提供可靠传输保障。由于主动权完全掌握在TCP/IP手中，所以TCP/IP向以太网提出了一个非常过分的要求，即以太网要想占有市场一席之地，就必须将MAC地址隐藏掉，对外统统用TCP/IP家族的新一代地址：IP地址。以太网委曲求全，不得不同意这个要求。

其实，TCP/IP提出这个要求不是故意难为以太网。因为除了以太网之外，还有很多其他类型的联网方式。而以TCP/IP在业界的权威性，其他网络都求助于TCP/IP来将它们融入市场以分一杯羹。而几乎每种联网方式都有自己的编址和寻址方式，面对这么多种地址，TCP/IP只好快刀斩乱麻，将这些五花八门的地址，统统映射到IP地址上，对外统一以IP地址作为编址方式。

下面简单介绍一下ARP协议。

既然TCP/IP宣称要将所有类型的地址全部统一到IP地址，那么也就意味着，网络中的每个节点，都必须配备一个IP地址，节点之间相互通信的时候，也要使用IP地址。但是，数据帧最终是通过底层的网络传输设备来转发的，如以太网交换机。而以太网交换机是不理解IP地址的，它只能分析数据帧中特定偏移处的MAC地址，从而做出转发动作。所以，必须有一种机制，来将IP地址映射成底层MAC地址。

ARP协议就是专门用来处理一种地址与另一种地址之间相互映射的一种机制。ARP协议运行在每个网络设备上，将一种地址映射成底层网络设备所使用的另一种地址。

对于ARP协议，本书不再作具体描述。

11.4.2 IP的另外一个作用

IP层还有一个作用，就是适配上下层，给链路层和传输层提供适配。适配了什么，怎么适配的呢？我们知道链路层有MTU的概念，也就是链路最大传输单元，即每帧所允许包含的最大字节数。

提示： 第7章中描述过，链路层就类似于司机和交通规则，要对货车的载重、车型大小有要求，如这条路承受不了多重的车，每辆车不能拉超过多高的货物等。如果客户给的货物太大、太重，不能一次运过去，那么只能把货物分割、分次运送，到达目的地后，再组装起来。这个动作由IP层程序来做。也就是IP根据链路的MTU值来分割货物，然后给每个分割的货物块贴上源和目的IP地址、顺序号，以便在货物块到达目的地后，利用顺序号来重新合并成完整的一件货物。

货物被分割成块后都需要被路由转发一次，然而路由器每次选择的

路径不一定都一样，而且每个块都需要由司机运送，司机驾驶水平、速度不同（链路层），就难免会有些先到、有些后到，所以到达目的地后很有可能乱序。此时就要用到顺序号了，这个号码是根据货物被分割处相对整个货物起点的距离（Offset，偏移量）而制定的。根据这个号，等所有货物块到达目的之后，对方的IP程序就会根据这个号码将零散的块组装起来。

强调：每个货物块都携带IP头部，但是只有第一块携带TCP或者UDP头部，因为传输层头部是在应用数据之前的，IP分割的时候，一定会把传输层头部分割在第一块货物中。

11.4.3 TCP/IP协议中的TCP和UDP

TCP/IP协议其实包含了两个亚层，IP是第一个亚层，也就是用来统一底层网络地址和寻址的亚层。第二个亚层，就是TCP或者UDP，在逻辑上它们位于IP之上。

TCP的功能，就是维护复杂的状态机，保障发送方发出的每个数据包，都会被最终传送到接收方，如果发生严重错误，还会向上层应用反馈出错信息，从而保证应用层逻辑的无误和一致性。

而UDP的功能，则可以理解是TCP/IP对以太网的一种透传，即UDP是一个没有传输保障功能的亚层，除了UDP可以提供比以太网更方便的调用方式外，其他方面没有什么本质区别。也许是因为TCP/IP协议觉得TCP的逻辑太过复杂，所以提供了一种绕过TCP复杂逻辑而又比以太网更加方便调用的方式，即UDP。

TCP/IP协议向上层应用程序提供的调用接口称为Socket接口，即“插座”接口。这也体现了TCP/IP想让应用程序更为方便地使用网络，就像

将插头插入供电插座而接入电网一样使用计算机网络。

基于TCP/IP有很多应用层协议，这些协议必须依赖TCP/IP协议，比如：Ping、Trace、SNMP、Telnet、SMTP、FTP、HTTP等。这些应用程序，加上它们所依赖的IP、TCP和UDP，然后加上物理层链路层（以太网等），一并形成了TCP/IP协议簇。

1. TCP协议

第7章中说过，TCP就是一个押运员的角色，也就是由它把货物交给IP做调度的。货物最初是由应用程序来生成的，应用程序又调用Socket接口来向接收方发送这些货物。应用程序通知TCP/IP去特定的内存区域将数据拷贝到Socket的缓冲区，然后TCP再从缓冲区将数据通过IP层的分片后，从底层网络适配器发送到网络对端。

TCP通过MSS（Max Segment Size）来调整每次转给IP层的数据大小。而MSS的值完全取决于底层链路的MTU值。为了避免IP分片，MSS总是等于MTU值减掉IP头，再减掉TCP头之后的值。这样TCP发送给IP的数据，IP加入IP头之后，恰好就等于底层链路的MTU值，使得IP不需要分片。

强调： 既然货物的大小要匹配MTU，那为什么不直接让TCP把分割好的货物给IP呢？

TCP其实很想这么做，但是它很难做到，因为TCP是个端到端的协议，也就是说，只有通信的最终端点维护着TCP状态信息，途经的各个其他设备一概不知道。如果两个端点所处的局域网都是以太网，但是途经一段串口链路，假设串口MTU为576B，以太网MTU为1500B，那么

双方在TCP握手的时候会互相通告自己的MSS，因为是端到端协议，不关心途经的设备和链路，那么它们都认为自己和对方都处在以太网中，所以互相都通告自己的MSS值为1460B（最大分段大小，等于出口MTU减去IP头和TCP头的开销）。

这样，TCP给IP的货物大小就是： $1460 + \text{TCP头} = 1480\text{B}$ ，然后加上IP头传输出去。一旦这个数据包到达了串口链路，则串口链路两端的IP层必须根据串口链路的MTU，对（ $1480 + \text{串行链路协议头}$ ）字节的数据包进行分片，将数据包分成多个576B的数据帧（最后一帧大小可能小于576B），从而在串口链路上传输。所以TCP的MSS，在广域网传输时基本派不上用场。不过，有机制可以探测到途经链路上的最小MTU值，TCP参考这个链路最小MTU值所得出的MSS值，这时是很有价值的。

在Windows系统中，每块网卡的MTU大小其实是可调的，但是只能调节到比网卡所连接到的以太网交换设备所允许的最小MTU值还小才可以，如果调节到大于这个值，则会造成数据丢失以及不可知的莫名错误。Windows中是通过注册表中以下的键值来调节MTU大小的：

```
HKEY_LOCAL_MACHINE\SYSTEM\ControlSet003\Services\TCP/IP\Param
```

图11-4是一个TCP握手过程中，发起连接端在TCP头的option字段中给出了本地TCP的最大MSS值（本地网络适配器最大MTU值减去40B）。本例中由于MTU值被配置成了1300B而不是默认的1500B，所以造成TCP通告MSS值为1260B。

图11-4 TCP握手过程中的一个包

TCP将上层应用的数据完全当作字节流，不对其进行定界处理。TCP认为上层应用数据就是一连串的字节，它不认识字节的具体意义，

却可以任意分割这些字节，封装成货物进行传送，但是必须保证数据的排列顺序。

比如，上层应用要传输123456789这9个数字，TCP可以一次发送这9个字节的内容，也可以每次发送3个字节，分3次发完，或者，上层放到货仓中1、2、3三个数据，TCP有可能将这3个数据打包一次发送（nagle算法），TCP的这些动作，上层都是不知道的。但是，对于123456789这9个数字，如果接收端先收到1、2、3这三个数字，4~9还没有被接收到，而当只有应用程序完全接收到这9个数字才会认为数据有意义时，应该怎么办呢？

要解决这个问题，就得完全靠上层应用程序。比如，可以在一个消息头部增加一个定长字段，表示这个消息的长度，比如在123456789之前增加一个（9），变成（9）123456789，那么程序一旦接收到（9），就能判断出接下来要连续接收9个字节才有意义，如果某时刻只接收到了1、2、3，应用程序便会将数据缓存起来，然后等待后面的6个数字。

如果TCP从（9）处分割了数据流，怎么办？比如分割成“（”和“9）123456789”两段数据，那么此时接收方应用程序接收到“（”这个字符，就会不理解其意义，这样就要求应用程序必须有缓存，把TCP交上来的数据流放到缓存中，然后自行合并成有意义的数据后再作处理。

TCP把上层的数据看作一些无关联逻辑的数据流，它不会感知到消息与消息之间的定界符。消息和消息之间的界限需要完全由应用程序自行分析。接收端和发送端的TCP都保存一个缓冲区（货仓），发送和接收的数据都存放在货仓中，接收方TCP货仓中的数据，每次被应用程序一次性取出到应用程序自己的缓冲区，应用程序再从应用缓冲中将数据流连接成有意义的数据进行处理。

一句话，TCP是把上层数据“分段”，IP是把TCP分好的段再“分片”（如果这个段大于MTU），IP到达目的之后会把每个分片合并成一个TCP的“分段”，提交给TCP，然后TCP就直接存放到货仓，顺序排放，不管上层消息间的分界。

2. UDP协议

UDP和TCP不同，TCP是孙悟空，那么UDP就是猪八戒。UDP只是被动地起到一个IP和上三层之间的接口作用，UDP没有传输保障机制，出错后不会重传，不需要保持重传缓冲和复杂的定时器、状态机等机制，而且UDP也不会像TCP那样把数据流按照MSS分段，UDP统统不理睬。用户传给UDP多大的数据，它就一次性发送出去，适配MTU的工作完全由IP来做。

UDP没有握手机制，想发就发，发完就不管了。正因为UDP这么简单，所以UDP头部只有8字节长，包括目的和源端口号、UDP数据包长度、UDP校验和。而且UDP相对TCP效率高了很多。所以它适合用在实时性要求很高，但是可靠性要求不高的时候，比如实时视频流、音频流服务等。

3. 端口号

计算机操作系统上，运行着N个程序，也可以说是N个进程。如果程序之间需要相互通信，就需要用号码来标识各个程序，也就有了进程号的概念。只要知道一个程序的进程号，就可以用这个区分其他程序。

同一台计算机上程序之间的通信，一般是在内存中直接通信的。如果两台计算机上的两个程序之间需要通信，虽然也可以通过高速网络将

两台计算机的内存共享，但是这种网络的成本很高。而普通网络能做到的，只能用另外一种方式，即先将消息通过网络发送到另一个计算机，然后让接收到消息的计算机来选择把这个消息发给其上运行的对应的程序。

要这样做，接收方的操作系统中的TCP/IP协议就一定需要知道某个数据包应该放入哪个应用程序的缓冲区，因为同一时刻可能有多个应用调用Socket进行数据收发操作。为了区分开正在调用Socket的不同应用程序，TCP/IP协议规定了端口号的概念。任何一个应用程序在调用Socket的时候，必须声明连接目的计算机上TCP/IP协议的那个端口号。

11.5 TCP/IP和以太网的关系

强调： 很多人把TCP/IP和以太网硬性关联起来，认为TCP/IP就是以太网，或者以太网就是TCP/IP，这种思想是完全错误的。

TCP/IP是一套协议体系，以太网也是一套协议体系，它们之间是相互利用的关系，而不是相互依存的关系。

TCP/IP协议并不像以太网一样有其底层专门的硬件，但是它可以租用一切合适的硬件来为它充当物理层和链路层的角色。除了以太网交换机，TCP/IP甚至可以用无线电波、红外线、USB、COM串口、ATM等作为其物理层和链路层。

以太网给TCP/IP充当了链路层，不一定代表它只能作用于链路层。以太网有自己的网络层编址和寻址机制，它有网络层的元素。各种联网协议都有自己的层次，都在OSI模型中有自己的定义，只不过TCP/IP协议在网络层和传输层的功能应用得太广泛了，所以OSI的第三层和第四层，几乎就是被TCP/IP协议给统治了，其他协议虽然也占有一席之地，但是相比TCP/IP的光辉就暗淡了许多。另外，TCP/IP没有统治链路层和物理层，在这两层中，就是其他协议体系的天下，所以TCP/IP只能“租用”其他底层协议（比如以太网），来完成OSI开放系统互联的任务。

这就是PoP，意即Protocol over Protocol。PoP的思想，到处可见，因为没有人可以统治OSI的全部7层，毕竟需要大家相互合作。

提示： 关于PoP的具体分析，在本书后面的章节中会讲到。

第12章 异军突起——存储网络的新军**IP SAN**

■ **TCP/IP**

■ 以太网

■ **SAN**

■ **PoP**

TCP/IP协议可谓出尽了风头，不仅统治了Internet，就连局域网通信，人们也愿意使用高开销的TCP/IP协议，可算是给足了它面子。

TCP/IP的买卖越做越大，吞并收购，不断涉足新领域，甚至连家用电器都想接入IP网络。而偏偏有一位愣是坚持不给TCP/IP面子，这就是大名鼎鼎的FC大侠。十几年来，TCP/IP在江湖上可谓是叱咤风云、前呼后拥、一呼百应，听惯了恭维话，看惯了鞠躬人。但是唯独FC大侠从来没正眼看过它一次，TCP/IP心里窝火啊！

12.1 横眉冷对——TCP/IP与FC

FC大侠既然敢和TCP/IP叫板，肯定有自己的拿手本领。

首先，FC在家底儿上就占了上风，FC是正儿八经的世家，四世同堂（OSI下四层都有定义）。而TCP/IP就两辈人（网络层和传输层），好不容易整了个后代还是收养的（租用以太网等其他底层传输网络）。

其次，FC目前普遍能跑出每秒4Gb/s的速度，以太网每秒100Mb/s的速度算正常，1Gb/s的速度算超常发挥，10Gb/s的速度还正在修炼之中。

再次，TCP/IP办事拖泥带水，笨重不堪。瞧瞧TCP/IP那个大头（TCP和IP头开销合起来要40B），下雨都能当伞用。FC的脑袋只有24B。以太网交换机MTU一般为1500B，FC交换机则超过2000B，传输效率上也高过以太网。

正由于这些原因，服务器和存储都愿意走FC的道儿。

老T（TCP/IP）心里一琢磨，虽然俺家业不如FC大，跑得不如它快，长得也比它胖，但俺也不是一无是处啊！

首先，俺广结天下良友，江湖各处都有俺的分号。

其次，俺便宜，给钱就让走。且俺的好兄弟以太网，几乎有网络的地方，一定少不了它。

再次，那FC也不是神仙，走它的道毛病太多，兼容性差，扩展性差，而且费用太高。就凭这三点，不信斗不过FC。

老T陷入了久久的沉思之中.....

12.2 自叹不如——为何不是以太网+TCP/IP

以太网可寻址容量很大，甚至比IP的地址容量都要大，是IP的 2^{16} 倍。而其地址是定长的，且使用专用电路完成交换动作。以太网除了双绞线之外，还可以用光纤进行传输。最重要的一点就是以太网非常廉价、部署简单。一个普通16口100Mb以太网交换机，只需要一两百元左右。而一个16口的FC交换机得上万元，还没有算上适配光纤的SFP适配器的费用。

但是，以太网与FC网络比起来，也有其先天不足之处。

第一，速度方面，以太网目前只普及到1Gb/s的速度，虽然10Gb/s以太网已经开发出了成品，但是离完全普及还需要一段时间。而FC已经普及到了4Gb/s的速度，且8Gb/s和10Gb/s速度的FC接口标准也正在制定当中。

第二，以太网是一个不可靠的网络，它不是一个端到端的协议，不管源和目的的状态，只是一味地向接口上塞数据，这也是下层协议的普遍特点。即使对方缓冲将满，以太网还是照样往链路上塞数据，而不会有所减慢。一旦接收方缓存充满，随后的数据帧就会被自动丢弃而不会向上层通告。所以，以太网必须依靠一种提供可靠传输机制的上层协议才能达到可靠传输。

强调： Disk SAN是唯一一个没有被以太网攻克的领域。最大的原因其实就是因为以太网的速度相对FC来说慢了太多。

以太网仿佛总是慢了一步。FC速率普及到1Gb/s的时候，以太网才刚刚普及到10Mb/s的速率，而且还是HUB总线式以太网。而当FC普及

到2Gb/s速率的时候，以太网也刚刚普及到100Mb/s。直到目前，以太网才普及到1Gb/s速率，而FC已经普及到4Gb/s了。

以太网即使是依靠了TCP/IP协议提供的传输保障机制，也难敌FC协议。总之，目前来说，FC协议在性能方面处处比TCP/IP+以太网强。

12.3 天生我才必有用——攻陷Disk SAN阵地

老T深知，自己再怎么修炼，也不可能在速度和性能上与FC正面交锋。在江湖中摸爬滚打了这么多年，老谋深算的老T决定避开自己的这些短处不谈，发扬自己的长处，将Disk SAN（相对于NAS SAN）这个阵地攻陷。

俺老T不管是论品相还是论才能，都不输给FC。既然SCSI能嫁给FC，它就没有理由对俺不动心。老T开始做白日梦，憧憬着与SCSI成亲后的种种。

老T经过一段时间的摸索和实验，终于设计出了一套新协议系统，称其为iSCSI，即Internet Small Computer System Interface。即在这种协议中，SCSI语言甚至可以通过Internet来传递，也就是承载于TCP/IP之上。

由此可见其扩展性是非常高的。只要IP可达，则两个节点之间就可以通过iSCSI通信。也就是说，位于中国的一台主机，可以通过iSCSI协议从Internet访问国外的存储空间。既然iSCSI协议是利用TCP/IP协议来传输SCSI语言指令，那么在通信的双方就一定需要先建立起TCP的连接。与FC协议类似，iSCSI将发起通信的一方称为Initiator，将被连接端称为Target。一般来说，Initiator端均为主机设备，Target端均为提供存储空间的设备，比如磁盘阵列。

老T拿着设计蓝图找到了SCSI协议，并且成功取得了SCSI的芳心。于2004年4月份完婚，并且领到了结婚证，编号为RFC3720。

两人并肩携手，成功游说了一批磁盘阵列生产厂商在其产品上尝试

着实现iSCSI协议。

提示： 正如本书第10章中所说的，既然SCSI语言及数据可以用FC协议传递，文件系统语言可以用以太网传递，那么SCSI语言当然也可以用以太网传递。

iSCSI既然要利用TCP/IP来传输SCSI协议指令和数据，那么就必须将自己作为调用TCP/IP这个传输管道的一个应用来看待。大家都知道在浏览器中输入“http://1.1.1.1”或者“http://1.1.1.1:80”，就表示让浏览器对IP地址为1.1.1.1的这台服务器上的TCP/IP传输管道的80号端口发起http请求，也就是浏览网页。同样，如果是telnet这个应用，那就要连接对方的23端口。所谓“端口”，就是被TCP/IP协议用来区分每个从管道中传出去或者收进来的数据包，到底是哪个上层应用的，哪个应用在“监听”某个端口，那么TCP/IP就将对应的数据包（数据包中含有端口号信息）发送到这个应用对应的缓冲区，正因为TCP/IP是个公用传输通道，谁都可以利用它来可靠地传输数据到网络另一端，所以才会用“端口号”来区分不同的发起数据传输的应用。

同样，iSCSI也要监听326这个端口号。SCSI指令和数据，作为“客人”，需要被TCP/IP这架“飞机”运载到目的地，SCSI本身并不关心也不想去关心诸如从哪个登机口（端口号）登机、行李托运、安检、海关交涉等一系列问题，于是需要有一个代理或者说引导者来完成这些动作，这个角色就是iSCSI Initiator和iSCSI Target。

早期的SCSI协议体系其实从物理层到应用层都有定义的，网络层也有定义，比如一条总线最大16个节点，每个节点有Target模式和Initiator模式，但是后期SCSI体系的下四层被其他协议取代，iSCSI+TCP/IP+以太网相当于取代了下四层，所以SCSI上三层不需要自己去发现网络里的

节点和Target了，这些都由iSCSI这个代理去完成。

首先iSCSI Target端运行在存储系统一侧或者说想要共享自己的磁盘空间让别人访问的那一侧，其作用是接收iSCSI Initiator端传输过来的SCSI协议指令和数据，并将这些指令和数据转交给自己这一侧的SCSI协议栈（其实是Class Driver或者SCSI Middle Layer）处理。iSCSI Initiator端运行在想要获得存储空间的主机一侧，其目的是向iSCSI Target端发起连接，并传输SCSI指令和数据。在iSCSI Initiator端程序中需要配置所要连接的iSCSI Target端的IP地址（或者使用一种叫做iSNS的服务动态自动配置来发现iSCSI Target），iSCSI Initiator会主动向这些IP地址的3260端口号发起iSCSI Login过程（注意这个动作不是SCSI协议定义的，完全是iSCSI这个代理程序自己定义和发起）。Login过程的交互细节此处不做细表，iSCSI Target端响应Login之后，双方在iSCSI层就连通了。iSCSI连通之后，Initiator会主动向Target发起一个SCSI Report Lun指令，Target便向Initiator报告所有的Lun信息，拿到Lun列表之后，Initiator端主动发起SCSI Inquiry Lun指令查询每个Lun的属性，比如设备类型（磁盘、磁带、光驱、打印机等等）和厂商之类，然后Initiator端便向OS内核注册这些Lun（这里要注意一下，这两条SCSI指令是Initiator固化的，不需要经过其上层的SCSI层）。OS内核便针对每个Lun加载其各自的驱动（Windows下就是Class Driver，Linux下就是Block Driver/Tape Driver等），在对应的/dev/下生成各自的设备。所以，iSCSI Initiator其实是一个虚拟的Port Driver，其通过调用TCP/IP，TCP/IP再继续调用底层网卡的Port Driver实现数据发送。

FC也是一个网络，也是替代了传统SCSI协议栈的下四层，FC也不是为了专门承载SCSI协议才被发明的，那么利用FC网络发送SCSI协议的那个应用程序或者说角色是什么？就是俗称FCP的一个协议，相当于

FC体系下的一个应用，也就类似TCP/IP体系下的FTP、Telnet等。FTP要发起连接传文件，首先要向对方的TCP端口号21发起连接，同样，利用FC传输SCSI指令的FCP，同样也需要向对方FC Target端某个特定端口号发起连接，FC Target端的某个程序正在监听这个端口的一切动作。那为何主机端不需要安装FCP Initiator程序呢？其实FCP的Initiator程序就是集成在了FC适配卡的驱动里了，因为FC的HBA卡目前来讲专门分配给存储用，所以直接集成到驱动里，不需要额外安装，而以太网则不同，厂商不可能自带iSCSI Initiator或者Target程序，所以一般都是独立开发独立安装。另外，FC协议也像TCP/IP协议一样有类似“端口号”的概念，只不过没有像TCP/IP这样被广为人知罢了，所以用FC承载任何上层应用都是可以的，当然，需要你自己去开发了。同样，SAS网络里也是这样一套运作流程，利用SAS网络承载SCSI协议，需要SSP发起端和目标端，同样，也被集成到了驱动里。

12.4 iSCSI交互过程简析

以下所有实例均为Windows XP操作系统环境，抓包软件为WireShark 0.99。

12.4.1 实例一：初始化磁盘过程

图12-1～图12-6所示的Trace结果是在Windows XP中，初始化一块通过iSCSI协议提交上来的LUN磁盘的过程中抓取的。在Windows初始化一块新磁盘的过程中，会对磁盘进行查询以及修改其MBR。下面就来分析一下具体的动作。

图12-1中包含了Frame1～Frame8。

图12-1 初始化磁盘的过程

关键帧分析

Frame1～8：ISCSI Initiator端（主机端）首先向LUN（ISCSI Target）发起Read Capacity指令来读取此LUN的容量信息。在Target返回的数据中，可以看到这个LUN共包含的LBA数为112454个，总容量为54MB（ $112454 \times 512 \div 1024 \div 1024$ ）。主机连续对Target发出了4次Read Capacity指令，这也是程序上的设计，可能是为了充分保证读取到的容量是准确无误的。

图12-2包含了Frame9～Frame10。

图12-2 Test Unit Ready

Frame9~10: 主机在读取完Target端的容量信息之后, 便发起了一个Test Unit Ready指令来探寻Target端是否处于可工作状态或已经准备就绪。第10帧是Target对主机的响应。

图12-3包含了Frame11~Frame13。

图12-3 Mode Sense

Frame11~13: 主机向Target发出了Mode Sense指令, 用来查询Target在iSCSI处理逻辑以及物理上的相关参数。Target在第12帧中返回了结果。

Frame14为TCP底层的ACK, 不必深究。

图12-4中包含了Frame15~Frame16。

图12-4 读取LBA0

Frame15~16: 主机读取Target的第一个LBA, 即编号为全0的LBA, 这个LBA也就是MBR扇区。从数据内容中可以看出, 新磁盘的这个扇区的内容是全0。

图12-5中是Frame17。

图12-5 写入MBR

Frame17: 主机向Target的0号LBA写入了数据, 也就是将一些必要信息写入了MBR。

图12-6中包含了Frame18~Frame57。

图12-6 多种指令再次读写

Frame18~57：在写入MBR之后，主机又接连多次发出Read LBA0、Read Capacity、Write LBA0、Test Unit Ready指令。在最后一个Test Unit Ready指令发出并获得返回数据之后，磁盘初始化完毕。此时没有数据包交互了。

对磁盘分区、格式化等操作过程，这里就简单地介绍这么多。有兴趣的读者可以自行实验。

12.4.2 实例二：新建一个文本文档

在图12-7所示的10.128.134.107这台主机通过TCP/IP网络与10.128.133.60那台磁盘阵列通信，并在磁盘阵列的一个LUN中建立了一个新文本文件。图12-7所示为主机与磁盘阵列交互数据的过程。

图12-7 一次写IO

前7个数据帧所包含的Payload字节数为8240。这8240个字节，就是主机向磁盘阵列所发送的一次写IO。协议分析软件自行判断了数据帧中的协议，并定界分析了前7个数据帧为一次iSCSI的写动作。这8240个字节，其实就是iSCSI协议模块通过一次Socket调用TCP/IP协议向外发送的字节，而TCP根据MSS值，又将8240B的数据分割成了7个数据包传送出去。这8240B中包含一个48B长的iSCSI头，以及剩余部分最终都需要被写入LUN的数据。我们可以计算一下，8240B减去48B，等于8192B，恰好是16个磁盘扇区（ $16 \times 512\text{B}$ ）的大小。也就是说，这次传输其实就是主机向这个LUN的16个连续扇区写入了数据。图12-8中也显示了相关字段的值，确实为16。

图12-8 此IO块大小为16个LBA

如图12-8在主机向存储设备传送的SCSI命令中，Opcode字段给出了这个命令的操作代码，0x2A表示写，如果为读，则对应的值为0x28。同时也给出了要写入的初始LBA为（也就是扇区号）198484和Transfer Length为16。也就是说，主机在这条命令中通知存储设备，将随后传送的数据写入从198484号LBA开始的随后16个扇区中。

存储设备上的TCP程序返回给主机3个ACK应答开销数据包后，在第11帧中，将iSCSI协议自身的会话层应答返回给主机，证明SCSI命令已经执行成功，如图12-9所示。

图12-9 成功返回

我们来计算一下主机向存储设备发送了16个扇区的数据，其所耗费的开销大致为多少。

开销1：iSCSI头部。iSCSI头部长48B，每次IO都只耗费48B。

开销2：TCP/IP头部以及ACK开销。在MSS值与MTU值适配的情况下，每个数据帧均会包含40B的开销；MSS值大于MTU值的情况下，IP会将数据包拆分，第一个拆分包耗费40B开销，随后的拆分包每个只耗费20B开销。ACK包的数量视窗口大小以及两端协议状态而定，数量不定。每个纯ACK包耗费60B开销。

开销3：以太网帧头部开销。每个以太网帧耗费14B开销。

根据以上描述，上述IO共耗费的开销为606B。所以，大致估算出的开销比例为 $606 / (606 + 8192) = 6\%$ 。当然这个估算是符合统计学原理的，在这里只是大致估算一下而已。

紧接着，主机又向存储设备发出了写入请求，这次写入的则是8个扇区，如图12-10所示。

图12-10 写请求

猜测：第一次主机向LUN写入了16个扇区的内容，是这个新建文本文件的实际数据，而第二次写入8个扇区的内容，可能是针对这个文件的元数据。当然在此只能做一个大致的猜测。若想追究到底也并不难，但是我们在此就不做过多分析了。

如图12-11所示是在向某LUN复制一个大文件时抓取的数据包。主机每次IO请求写入的扇区数变成了128，也就是64KB的数据。从数据包内容中，可以看出这个文件好像是一个视频文件，而且是一部流行的美国电视剧。

图12-11 复制一个大文件

由于Socket的缓冲区一般为64KB，所以上层程序每次调用发送给Socket最大的数据长度也被限制在64KB。由此就可以计算出图12-11中的这个IO将数据写入了LBA305768~LBA305895这128个扇区中。如果此时还需要接着写入新数据而同时LBA305895之后的扇区还是空闲状态的话，文件系统便会继续向其后的扇区继续写入。所以，图12-12中所示的IO向LBA305896~LBA306023这16个扇区写入了64KB的数据。如果文件依然没有写完，则继续按照这个逻辑写下去。

图12-12 连续的64KB写入

12.4.3 实例三：文件系统位图

图12-13～图12-15所示的Trace结果是在通过主机上的文件系统，将500MB的数据从一个写满数据的1GB iSCSI LUN中删除的过程中所抓取的。

图12-13 删除500MB的数据

图12-13为删除500MB数据时抓取的数据包。

由于文件系统从磁盘上删除数据的过程中，只会修改相关的链表，从元数据中抹掉相应的记录，而不会去抹掉或者覆盖被删除的文件原来所对应的扇区上的任何数据，所以虽然删除了500MB的数据，但是真正的IO数据远小于500MB。本例中，这个过程只交互了163个数据包。

在前51个数据包中，所有IO均为写操作，每个大小均为8个扇区（4096B，被分成三个数据帧）。可以判定这些IO其实都是在更新文件系统元数据。

从第52～第80个数据包为一个80扇区的写IO，如图12-14所示。

图12-14 文件系统位图（一）

分析这个IO的实际内容，发现其包含了很多的0xff，即二进制的11111111。这个IO更新的元数据，就是本书前文中提到过的文件系统位图（见5.7.4节）。位图是一个元数据文件，其中用每个位来表示对应磁盘分区中的每个块（或者簇，视设计不同而定）是否正在被某个文件所占用。由于本例中，LUN上依然留有一半的数据，所以依然有一半的簇被文件占用着。这些簇也就是图中被标明为0xff字节所对应的簇。被标明0x00的字节则对应着磁盘上未被文件占用的空闲块（簇），但不是说这些空闲块中“没有数据”或者“数据为全0”。对于一块崭新的磁盘，扇

区中的数据可能为全0，但是对于一块已经使用过的磁盘，扇区中会保留很多以前被删除文件的“尸体”。这些“尸体”会被随后的对这个扇区的写IO数据所覆盖。

图12-15所示的数据是在将那个LUN中剩余的数据全部删除的过程中所抓取的。

图12-15 文件系统位图（二）

可以看出，LBA1044425以及随后（包括其自身）的56个扇区为存放位图的扇区（当然这只是一部分，还可能还有其他扇区也用来存放位图）。由于删除文件导致LUN上的空闲块增加，所以文件系统必须修改位图映射表，这就产生了图12-15中第246个数据包所对应的写IO操作。通过数据包内容可以看出，位图中几乎所有字节都为0x00。这就说明此时LUN上已经几乎都是空闲块了。

读者可以自行操作并分析一下iSCSI以及文件系统的逻辑。但是要注意一点，由于主机上的文件系统是有缓存的，当向LUN做了一次操作之后，会被文件系统缓存一段时间（几秒或者十几秒都有可能），然后才批量Flush到硬盘（LUN）上。所以此时一定要保持网卡抓包软件继续执行抓取，直到十几秒钟之后，流量面板中没有新的数据包被抓取到，此时方可停止抓取，获得Trace数据。

12.5 iSCSI磁盘阵列

当年FC闹革命的时候，可谓是轰轰烈烈，气壮山河。当时，那场革命对磁盘阵列架构的改变真是太彻底了，所以至今人们仍然记忆犹新。而TCP/IP想在已经被FC革命过的江湖上再闹出点动静来，可就不是那么容易了。毕竟TCP/IP所依靠的以太网相对于FC来说，都属于包交换网络，而且很多概念都类似甚至相同。很多盘阵厂商并不看好TCP/IP的介入。但是最终还是有一批喜欢标新立异、敢于创新的厂商，举起大旗来支持TCP/IP。

图12-16便是一个典型的iSCSI磁盘阵列的基本架构。可以看到，其前端IO设备就是普通的以太网卡。TCP/IP以及iSCSI逻辑均运行在主内存中。后端的磁盘可以以任何方式接入总线，甚至可以是独立的磁盘阵列。磁盘经过VM虚拟化层之后，通过前端接口供主机访问。iSCSI盘阵与FC盘阵结构类似，只不过前端接口成了以太网口而已。NetApp的FAS系列产品在同一个控制器上实现了多种协议的接口，包括iSCSI、FCP、CIFS、NFS以及其他的一些基于TCP/IP的数据访问协议。

图12-16 iSCSI盘阵架构示意图

注意： VM意为Volume Manager， Virtualization Manager。

我们可以发现，这台盘阵的架构与PC无异。的确，不管是主机还是磁盘阵列，它们都是由计算机系统的老三样——CPU、内存、外设组成的。其所实现的功能，关键是靠所运行的软件。主机上的老三样运行的是处理业务逻辑的应用程序；而运行在盘阵老三样上的则是专门处理通过不同网络协议传输进来，或者出去的SCSI指令以及优化磁盘读写的程序。从本质上来说，它们二者是相同的，只不过是分工不同而已。

提示： 正因为它们本是同根生，所以相煎又何太急呢！PC只要运行了盘阵上的软件，就是一台盘阵；相反，盘阵如果运行了应用程序，则就可以当成一台主机来使用。

目前，几乎各种操作系统都已经有了iSCSI Initiator软件。有些操作系统，比如Windows、Linux等还有了iSCSI Target软件。它们安装了Target软件，也就变成了盘阵，只不过在性能、功能和容量上没有专业盘阵强悍。

目前，TCP/IP只是占领了盘阵前端接口的部分阵地。而对于后端磁盘接口的进攻，也不是没有设计和尝试过。当年FC可是一举拿下了盘阵的前端和后端。而如今老T能有FC的那个本事么？确实有些厂家的盘阵将TCP/IP协议作为后端磁盘到适配器之间的传输协议，但是这样的设计似乎并没有得到认可。这个结果也是可以预知的，TCP/IP之所以可以与FC竞争，就是因为其优良的扩展性，而不是因为它的速度。后端需要的首先是性能，而不是扩展性。所以，后端还是乖乖地交给FC才是明智的选择。

12.6 IP SAN

后来，人们索性将iSCSI为代表的以TCP/IP作为传输方式的网络存储系统称做IP SAN，即基于IP的存储区域网络。值得说明的是，IP SAN并不一定要用以太网作为链路层，可以用任何支持IP的链路层，比如ATM（IPoA）、PPP、HDLC，甚至是Fibre Channel也可以作为IP的链路层。

这样，就使得IP SAN的可扩展性变成了无限，它可以扩展到世界上任何一个有Internet网络接入的地方。这也是Internet Small Computer System Interface名称的由来。

iSCSI的方便和灵活性逐渐显现出了优势。的确，现在还有哪台主机上不带以太网适配器的？还有哪台主机上不运行TCP/IP协议的呢？就连大型机设备都有自己的前置TCP/IP处理机了。

FC网络虽然比并行SCSI总线的扩展性高了很多，但是相对于TCP/IP的扩展性，FC就是小巫见大巫了。

iSCSI与NAS的区别如下。

虽然iSCSI与NAS都是利用TCP/IP+以太网来实现的，但是二者所传输的语言是大相径庭的。NAS传输的是文件系统语言，而iSCSI传输的是SCSI指令语言。NAS设备上必须运行一种或者多种文件系统逻辑，才能称为NAS；而iSCSI Target设备上不需要运行任何文件系统逻辑（盘阵自身操作系统文件管理除外）。

在相同的条件下，iSCSI与NAS在速度与性能方面相差不大。

12.7 增强以太网和TCP/IP的性能

老T对IP SAN可谓是投入了一腔热血。它想尽一切办法要提高TCP/IP的性能，以便与FC抗衡。

1. Checksum Offload (CO)

计算每个TCP包的校验数据是一件极其枯燥乏味和耗费资源的工作。由于TCP/IP程序均需要运行在主机操作系统中，所以计算校验数据的任务当然要落在主机CPU身上。CPU不得不拿出额外的指令周期来计算每个TCP包的校验数据。这对于CPU处理能力比较弱的主机来说，性能影响是不可忽略的。

为了将CPU解脱出来，一种称为Checksum Offload的技术被开发出来。这种技术将计算校验数据的工作完全转移到了网卡的硬件上，对于向外发送的TCP包，CPU可以不经校验直接传送给网卡，由网卡芯片来做校验；同样，对于接收到的数据包，在没提交到主机内存之前，就已经做好了校验匹配。图12-17就是Checksum Offload功能的一个图示。

图12-17 网卡的Checksum Offload 功能

2. Large Send Offload (LSO)

TCP需要根据底层链路的MTU值来适配其每次发送的数据大小。如果上层传递过来的数据大于其允许的最大分段长度（MSS值），则TCP会将这些数据分成若干个数据包发送出去，这就是所谓的Large Send。这项工作目前也可以被转移到网卡上来完成。

3. TCP/IP Offload (TO)

TCP/IP Offload则干脆将TCP/IP整个程序都放到网卡硬件芯片上来运行。这种特殊的以太网卡称为TOE卡，即TCP/IP Offload Engine Card。图12-18是TOE卡在存储盘阵上的应用及架构图。

图12-18 使用TOE卡的iSCSI盘阵层次架构示意图

4. Security Offload (SO)

Security Offload不仅将TCP/IP协议从主机上Offload了下来，它还可以在硬件上直接实现IPSEC相关的协议，将对数据包的加解密过程也从主机上Offload下来。

5. iSCSI Offload (IO)

iSCSI Offload将TCP/IP+iSCSI的整套逻辑都放到网络适配卡上来运行。由于iSCSI的上层是SCSI，所以一张iSCSI卡对于主机来说，会表现为一张SCSI卡。不同的是，这张虚拟的SCSI卡可以设置其自己独立的IP地址以及其他TCP/IP和iSCSI的参数。图12-19所示为iSCSI硬卡在存储阵列上的应用及架构图。

图12-19 使用iSCSI Offload卡的iSCSI盘阵架构示意图

12.8 FC SAN节节败退

1. 成本问题

说来有点惊讶，部署FC SAN的成本是部署IP SAN的10倍甚至十几倍。以太网卡和以太网交换机相比FC卡与FC交换机便宜了很多。所以，对于对性能要求不是很苛刻的用户来说，部署IP SAN无疑是性价比最高的方法。

2. 可扩展性问题

FC是一个专用网络。虽然FC当初是作为像以太网一样的通用的网络传输技术被设计出来，但是目前，其专门被用作存储网络，也算是命该如此。FC长期被束缚在这样一个狭小的环境内，不仅造成了其不思进取的性格，而且也造成了其成本的居高不下，所以FC很难被扩展出去。

3. 易用性问题

曲高则和寡。既然FC这么不开放，那么就注定难用。部署一个FC存储网络比部署一个IP存储网络要复杂，对技术人员的要求也比较高。

4. 兼容性问题

由于FC极其不开放，即使FC有相关的标准，但不同的生产厂家生产出的FC设备，有时候并不一定会完全兼容，总会出现一些莫名其妙的问题。相比来说，TCP/IP由于已经在完全开放的环境中摸爬滚打了很

长的时间，所有已经被发现的bug也都被修复了。

12.9 iSCSI配置应用实例

本实例用一台NetApp FAS3050系列磁盘阵列充当iSCSI的Target设备，用一台运行于Windows XP操作系统的PC充当iSCSI Initiator端，PC上的iSCSI Initiator软件为微软的MS iSCSI Initiator 2.07版本。本例中将描述如何在存储设备上一步步地创建LUN，然后映射给主机使用。

12.9.1 第一步：在存储设备上创建LUN

1. 创建Aggregate

所谓Aggregate是指RaidGroup的组合，一个Aggr（Aggregate）可以包含多个RG（Raid Group）。如图12-20所示，这台盘阵在其后端的0a和0b个FC通道下各连接了两台扩展柜（Shelf1和Shelf2），每台扩展柜包含14块硬盘，这样每个通道包含了28块硬盘。

图12-20 系统硬盘列表

图12-21所示的是系统中目前还没有被分配到RG中的磁盘，标为Spare，共16块。

图12-21 Spare硬盘列表

使用“aggr create”命令创建一个aggr，如图12-22所示。“-r 6”参数表示每个RG最大允许包含6块磁盘，如果超过6块，则自动形成另一个RG；“-t raid_dp”参数表示RG的RAID类型为RAIDDP（这种RAID类型本书不做描述，具体可以在Internet上搜索）；12表示这个aggr包含12块盘。计算一下这样正好可以形成两个RG，每个RG6块盘。“aggrtest”为

这个aggr的名字。

图12-22 创建aggr

创建好的aggr中有两个分别包含6块盘的RG，如图12-23所示。

图12-23 aggrtest中包含两个RG

刚创建好的aggr名为aggrtest，大小为955GB，其中给快照预留的空间为47GB，实际可用在线存储用户数据的空间为908GB，如图12-24所示。

图12-24 aggr的空间分布

2. 在aggr中创建Vol

Vol，即卷，是凌驾于aggr之上的一层虚拟化产物，目的是为了更加灵活地取用aggr所提供的存储空间。Vol可以任意创建删除，任意增加或者减小容量。

如图12-25所示，在aggrtest中创建了一个名为voltest，然后将offline删除。这个过程非常简单，命令发出之后立即生效。

图12-25 创建/删除一个Vol

在aggrtest上创建两个卷：iSCSI1和iSCSI2，大小均为300GB，如图12-26所示。

图12-26 创建两个新卷

3. 在Vol中创建LUN

LUN是最终提交给主机使用的一块存储空间，NetApp将LUN容纳于Vol的空间之下，并且LUN也可以任意增、删、扩、缩。

在每个Vol中分别创建一个大小为200GB的LUN，如图12-27所示。

图12-27 创建LUN

4. 创建Igroup并映射LUN

所谓Igroup，是用来管理LUN—主机映射关系的。本书前文中描述过这种灵活的映射关系。Igroup就像一个桥梁，主机和LUN如果都映射到某个Igroup，那么这台主机就可以访问这些LUN。

如图12-28所示，创建一个名为IG的iSCSI类型的Igroup，并将它映射到了一个主机端的iSCSI Initiator的IQN地址。接着，将两个LUN都映射到了这个Igroup中。

图12-28 创建Igroup并映射LUN和主机

12.9.2 第二步：在主机端挂载LUN

1. 确认主机端的IQN名称

确认主机端iSCSI Initiator的IQN名称是否与Igroup中所配置的一致，如图12-29所示。

图12-29 IQN名称

2. 添加iSCSI Target端地址

添加iSCSI Target端地址，发现其上的LUN，如图12-30和图12-31所示。图12-32显示已经发现iSCSI目标，但处于未激活状态。

图12-30 添加iSCSI Target端地址

图12-31 添加完成

图12-32 未激活状态的Targe

3. 激活iSCSI目标端，发现LUN

单击图12-32中的Log On按钮，如图12-33所示。

图12-33 激活iSCSI目标

单击OK按钮后，可以在图12-34中看到，目标已经连接，此时LUN会被主机识别到。

图12-34 处于激活状态的iSCSI目标

4. 使用LUN

如图12-35所示，在主机的磁盘管理器中会发现两块未初始化的磁盘，其对应的是存储设备上的两个LUN。

图12-35 新识别到的两个LUN

初始化之后格式化这两块磁盘（LUN），如图12-36所示。

图12-36 初始化之后格式化磁盘

如图12-37和图12-38所示，这两块磁盘已经可以正常使用了。

图12-37 磁盘状况正常

图12-38 “我的电脑”显示的磁盘

12.10 iSCSI卡Boot配置示例

上文提过iSCSI卡，即向操作系统表现为一块SCSI卡，底层却用IP网络来传输SCSI协议和数据。本节介绍如何配置系统从iSCSI启动。在阅读本节之前最好先阅读9.9节，本节介绍的内容与FC Boot过程类似。

如图12-39所示，系统加电后进入iSCSI卡的Optional ROM配置界面。首先配置iSCSI Initiator端的信息，比如IP地址、网关地址等。配置完后进入Target端配置界面。

图12-39 配置iSCSI Initiator端信息

配置Target端的IP地址和IQN名，如图12-40所示。配置完毕后退出现。在系统主Bios中配置已经识别到的iSCSI LUN作为第一启动盘，如图12-41所示。

图12-40 配置iSCSI Target端信息I

图12-41 主Bios中配置第一启动盘I

图12-42所示为Intel Pro 1000/PT iSCSI卡的配置界面。

图12-42 Intel Pro 1000/PT iSCSI卡的配置界面

当iSCSI卡配置完成之后，系统重启的过程中会出现类似图12-43所示的界面。

图12-43 配置完毕系统重启之后的界面

12.11 10Gb以太网的威力初显

2010年3月，Microsoft与Intel合作进行了一项在10Gb/s以太网之上基于iSCSI协议的IOPS及吞吐量测试。测试中硬件使用基于Intel Xeon processor 5580 CPU的服务器，以及基于Intel 82599 10 Gigabit Ethernet Controller的10Gb/s以太网适配器；操作系统可以用Windows Server 2008 R2，以及Microsoft iSCSI Initiator软件；存储端使用StarWind公司的iSCSI Target软件，交换机使用Cisco Nexus 5020 10Gb/s以太网交换机。整个系统中使用10台iSCSI Target Server和1台iSCSI Client，Client使用单10Gb/s以太网口连接交换机。在512Byte IO Size和连续IO条件下，获得了双向总共超过100万的IOPS值。如图12-44所示分别为IOPS和吞吐量（双向）。

图12-44 10Gb/s以太网的IOPS和吞吐量

当然，连续小块IO是最能够获得高IOPS的IO方式，并且测试使用的硬件和软件都是做了充分优化的，比如Intel® 82599适配器开启了众多的Offload，包括Checksum、Large Send等，以及针对Intel Xeon5000系列CPU的一些额外优化，比如在中断和DMA方面。

这个测试的目的是为了饱和链路的IOPS，证明了10Gb/s以太网+软iSCSI Initiator在经过优化设置之后，完全有能力饱和底层链路的IOPS和带宽。目前业界对TCP/IP+以太网组合的协议效率有很多质疑，质疑其效率非常低，有人甚至说效率只有50%，这完全是无稽之谈。

随着整个网络骨干速度的提升，以太网取代不开放的FC平台，是迟早的事情。

12.12 小结

IP SAN出现之后，FC SAN的统治地位被大大地动摇了。FC目前唯一可以用来与IP SAN抗衡的武器，就是其高速度。但是这个武器的震慑力也在降低。我们可以发现，以太网的速度每次革新都是以10倍速为单位，从10Mb/s到100Mb/s、1Gb/s，10Gb/s速率的以太网也已经发布了。而FC每次革新均以2倍速为单位，从一开始的1Gb/s、2Gb/s，到目前正在被广泛使用的4Gb/s速率，而FC的下一个速率级别为8Gb/s（或者10Gb/s），而且这个标准还正在制定当中。以太网已经在计划上抢先了FC一步。不知道在10Gb/s时代，FC还能不能守住它那仅有的一点点沙漠绿洲。

说明：现在已经出现了HyperSCSI以及ATA over Ethernet这两种欲抛弃略显低效的TCP/IP协议而另立门派的协议，但是由于各种原因，尚未得到推广。

第13章 握手言和——IP与FC融合的结果

■ FC

■ IP

■ 协议之间的相互作用

■ 协议融合

话说FC和IP 各占一方，谁也不让谁，互相竞争了数年，两者各立门派，势不两立。但是“夫天下之势，分久必合，合久必分”。

数年来，两者在市场上竞争得可谓你死我活。FC仅仅拿着FC SAN的速度和稳定性来炮轰IP SAN，而IP 也不甘示弱，处处举着可扩展性和成本的大旗，声讨FC SAN，闹得江湖上风风雨雨。FC凭借着它的速度优势，占据了高端市场，而IP 则以成本优势在低端市场占据了一席之地。然而两人谁都想一统天下，把对方彻底驱逐出市场，但是，相持了数年，谁也没能把谁干掉，两人都累了。这么多年的互相攻击，谁也没有取得丝毫胜利，FC还是稳固地占据高端市场，IP 依然驰骋低端。

终于有一天，FC和IP 决定握手言和，不再投入无谓的人力、物力、财力来和对方竞争。与其大肆攻击对方，不如多用点精力来提升和发展自身的技术，同时学习对方的技术，取长补短，方为正道啊！FC和IP 彻夜长谈，终于取得了一致的见解，决定双方各取所长，共同为江湖做贡献。

首先，FC决定由IP 入股自己的公司，给FC SAN提供更高的扩展性架构解决方案；同时，FC也入股IP 的公司，给IP 提供研发经费，用于其研发出基于以太网的、新型的、适合存储区域网络的专用协议体系。

13.1 FC的窘境

入股FC公司之后，IP 便开始研究如何将FC协议体系转向可扩展的、开放的结构。说到可扩展且开放的网络传输协议，一定非TCP/IP莫属。可是FC和TCP/IP是两套毫不相干的协议体系，如果将FC全部转为TCP/IP，那岂不是叛变成IP SAN了么？但是如果丝毫不变，那只能是FC SAN，还是不具备开放和扩展性。

1. FC的扩展性问题

FC为什么扩展性差？就是因为如果通信双方距离太远的话，需要自己架设光缆，或者租用电信的专线光缆，这两者成本都很高。如果租用电信部门的专线光缆，则FC最低速度为1Gb/s，且租用电信部门的1Gb/s带宽的专线光缆，其费用不是一般机构能承担的。

提示： 目前电信提供的专线接入，其骨干网一般采用PDH或者SDH协议传输，到终端用户所能承受的速率为2Mb/s的E1线路。当然也可以直接从高速骨干网直接分离出相对高速的线路，比如OC3、OC48等，但是费用还是过于高昂，令人无法承受。

E1线路有自己的编码格式，不能将电信部门接入的光纤直接插到FC设备上，因为两端的编码方式不同，不能和局端的设备建立连接，所以需要增加一个协议转换设备（准确来说是协议隧道封装设备），将E1协议解封装，转换成协转设备后面的协议逻辑，比如V35串口、以太网等其他协议。目前已经存在FC over Sonet、FC over ATM等协转设备了，不过这些专线的扩展性仍然不强，而且这种方案以及对应的设备也

非常昂贵和稀少。

目前看来，如果要扩展FC网络，让相隔很远的两地之间用上FC协议，最好的办法就是自己架设专用光缆。可是自己架设光缆也只能在自己可控的范围内，比如一个大厂区之内，但是如果是在市内，或者两个城市、两个省之间，私自架设光缆是绝对被禁止的。

2. 解决方案

怎么办？首先，要走出去，就一定要租用电信部门的线路。电信提供了两种线路：一种是接到Internet的线路，也就是接入电信部门的Internet运营网络，通信的双方都接入，并且使用TCP/IP通信；另一种就是光缆专线，也就是通信的双方都接入电信部门的专用传输骨干网络，这条专线端到端的带宽由接入提供商保证，只要两端的设备支持，其上可以运行任何上层协议。上层帧会被底层封装协议（比如E1等）再成帧传送到电信部门骨干传输网络中。

虽然Internet接入可以获得100Mb/s或者1000Mb/s的速率，但是这只是本地带宽（从本地到局端设备之间的链路带宽），端到端的带宽，以现在的电信部门TCP/IP网络环境，除非购买接入商的QOS或者MPLS TE服务，否则没有人能够保证两点间的通路带宽（速率）。

提示： 如果两地之间相距很近，那么不妨考虑Internet链路。因为如果两地同时接入相同城市的ISP网络，数据包被路由的跳数就不会很高，甚至有可能只经过1跳或者2跳便可以被对方收到。更有甚者，同城的两地可能连接在局端的同一台设备上。这样可获得的带宽速率就会非常可观，就可以像在内网通信一样利用VPN来让两个站点之间联通。但要澄清一点，由于

Internet链路不能时刻保障稳定的带宽，所以这种方法只适合对数据传输实时性要求不高，但同时又要求高带宽的情况。

而专用线路虽然保证了带宽，但是只能承受E1等低速专线，且价格相对Internet接入要贵很多。而且目前只有V35—E1封装解封设备和E1—以太网封装解封设备，并没有E1—FC封装解封设备。而V35串口和以太网这两种二层协议，都普遍被用来承载IP协议，所以目前来说，E1一般用来承载IP作为网络层协议。有些路由器自带E1封装解封模块，可以不用外接协转，直接连接从光端机分离出来的一路或者几路G703或者BNC接头，直接编码与解码E1协议。但是这些也都是IP路由器，和FC丝毫没有关系。

可以看出，FC如果脱离了“后端专用”这四个字到开放领域，显然是无法生存的。而IP SAN，则软硬通吃，只要有IP的地方，不管其下层是什么链路协议，就可以部署IP SAN。这就是为何称TCP/IP为协议中的秦始皇的原因：秦始皇统一了货币，到哪里都通用，同样，TCP/IP也统一了下层凌乱的各种协议。

13.2 协议融合的迫切性

说到这里，租用Internet线路，只能承载IP，而租用点对点专线，也普遍用来承载IP，可能感觉FC的扩展似乎就是死路一条了。但是，IP想起了ISCSI，当初自己不就是把SCSI协议给封装到了TCP/IP协议中来传输，才扩展了SCSI协议么？也就是说如果将一种协议封装到另一种协议中传输，就可以使用另一种协议带来相应的好处了。不妨就这么假设一下，FC不可扩展，TCP/IP扩展性很强，那么如果把FC协议封装到TCP/IP协议中来传输，是不是也可以获得TCP/IP的扩展性呢？这个想法比较大胆，因为FC本身也是作为一种可以传输其他协议的协议，FC甚至可以承载IP，作为IP的链路层，那么为什么现在却反过头来需要用IP来承载呢？

提示： Protocol over Protocol, PoP，即一种协议被打包封装或者映射到另一种协议之上。这种思想在网络协议领域中经常使用。我们姑且称其为“协议融合”，认为其已经可以形成一个独立的科目。

要谈协议融合，还得从以太网和TCP/IP说起。

1. 以太网和TCP/IP——不能不说的故事

前面已经详细介绍了以太网和TCP/IP协议。我们知道，以太网是一个网络通信协议。

提示： 记得某人曾经说过一句话：“网络就是水晶头。”这句话比较有意思，它反映出说这句话的人对网络的不了解，但是也证明他平时所见到的网络，确实只有水晶头，且以太网普遍使

用水晶头，这样“网络就是水晶头”这句话，也不是那么可笑了。它从某种角度也反映出了以太网在当今的普及程度。

前面讲到以太网是可以寻址的，也就是说它涉及了OSI第三层网络层的内容。大家都连接到一个以太网环境中，不需要任何其他上层协议，就可以区分对方，进行通信。既然这样，为什么连新闻联播的主持人都知道Internet是利用TCP/IP协议而不是以太网来通信呢？为何我们总是说以太网+TCP/IP协议二元组，而不是仅仅说以太网，或者TCP/IP协议？

因为以太网和TCP/IP协议是逻辑上分开的，它们各自是不同的协议体系，那么为什么总是把它们组合起来说呢？它们之间有什么割舍不断的恩恩怨怨呢？这其中原因，还要从IP讲起。

2. IP本位

前面也说过了，IP就是一个身份标志，是用来与其他人区别的一个ID。以太网协议中规定的MAC地址，从原理上讲，就足够用来区分网络中各个节点了。但是前面也分析过，完全靠MAC来寻址的缺点：一是MAC地址太长，48b，用于路由寻址时效率太低；二是世界上并不是每个环境中都用以太网来建立网络，除了以太网，还有其他各种方式的网络系统，各自有各自的寻址方式，如果要让所有类型的网络之间无障碍的相互通信，就需要一个秦始皇来统一天下的货币。

IP就是这个被选中的货币。不管以太网，或者串口协议，或者FDDI等类的局域网方式，我们最终都要让其之间相互通信，才能形成Internet。

提示： 如果你是秦始皇，你会怎么来处理各国众多的货币呢？虽然秦始皇最终将其他货币回收废除了，但是IP却不能在短时间内将所有网络形式都废除，而用以太网统一，因为现在已经不是一个人说了算的时代了。秦始皇可以在各个使用不同货币的地方设立一个专门的兑换机构，只要到了这个地方，就兑换成这里使用的货币。

同样，我们也给每个网络设立一个网络地址兑换设备，也就是协议，将统一的IP地址兑换成这个网络的自用私有地址，用这种方式实现各种类型网络的相互联通。网络中的兑换机制，是通过ARP协议实现的，ARP协议可以将一种网络地址映射成另一种网络地址。每种网络要想用IP来统一，都必须运行各自的ARP协议，比如以太网中的ARP协议，帧中继网络中的ARP协议等。

对于以太网来说，IP就是统一货币，MAC就是以太网货币。另外，还有各种各样其他类型的货币，比如主机名（Hostname）、域名等。大家在访问网站的时候，其实就是和提供网站服务的服务器来建立通信，获取它的网页和其他服务，在IE浏览器中输入这个网站的域名之后，DNS兑换程序会自动向DNS服务器查询，获得这个域名所对应的IP地址，然后用IP地址与服务器通信。

数据包带着IP地址到了服务器所在的局域网之后，会通过局域网的路由器发出ARP请求，来把IP地址再兑换成服务器所在局域网络的地址。如果服务器所在的局域网是以太网，则对应成MAC地址，然后通过以太网交换设备，找到这个MAC地址所在的交换机端口，将数据包发向这个端口，从而被服务器收到。

为什么要经过多次兑换呢？首先把IP转换成域名，是为了方便记

忆，不必记忆那些复杂的IP地址；其次把MAC转换为IP，是为了天下统一，相互流通。

其实如果所有人都用以太网联网，那么就可以完全抛弃IP这一层寻址了，但是实际是不可能的，以太网现在还没有一统天下，而且就算一统天下了，人们也似乎不愿意抛弃IP，就像在同一个局域网内，还是用IP来直接通信，而不是直接用MAC。TCP/IP实在是被使用的已经太普遍了，以至于就算牺牲一点性能，局域网内通信也普遍使用IP。而实际上，以太局域网内部通信的话，NetBEUI协议的性能比TCP/IP协议要高许多。

其实整个Internet不仅仅都是以太网，以太网适合局域网联网通信，但是不适合广域网情况，广域网的联网协议，比如PPP、HDLC、Frame Relay、x25、ATM等，也像以太网一样各有各的寻址体系。在一个Internet上有这么多种不同地址的网络，它们之间若要相互融合、寻址，就必须在各种地址之间，相互翻译、转换、映射，数据包每经过一种网络，就转换一次，这样非常麻烦。IP地址的出现使得所有联网的节点，不管用的是以太网，还是Frame Relay，统统都分配一个IP地址给每个节点，对外最终以IP地址作为寻址地址，而将IP地址再映射到自己所在网络的所使用的地址上，比如IP映射到以太网的MAC，或者IP映射到Frame Relay的DLCI、映射到ATM的地址等。

用来进行地址映射的程序，称为Address Resolution Protocol，即ARP。很多人听到ARP，就认为是以太网，其实这也是错误的，ARP不仅仅代表以太网中的IP地址和MAC地址的映射，它代表任何种类地址之间的映射对应关系，从这一点来说，DNS协议也应该归入广义的ARP协议中。

IP统治了OSI的第三层，将原来占据第三层的凌乱地址种类统一了。映射到（承载于）以太网的IP，称为IPoE（IPoE也就是“基于以太网的TCP/IP”）；映射到帧中继的IP，称为IpoFR；映射到ATM的IP，称为IPoA等。从此一种新的概念诞生了：PoP，即Protocol over Protocol。

3. IP缺乏传输保障功能

IP统一了天下还不够，因为IP最大的作用就是寻址和路由以及适配链路层MTU，它并不提供其他功能，而作为一个健全的网络传输协议，必须具有传输保障功能。而以太网是一个面向无连接的网络，它不保障数据一定会传送的对方，是一个不负责任的网络，不管目的端口有没有收到，源端口只管向外发送。而Frame Relay协议，其前身x25协议，是一个有着很好传输保障功能的协议，在TCP/IP没有出现之前，x25的传输保障机制做得非常到位，因为x25的设计初衷，就是为了运行到极其不稳定的链路上。而随着链路质量的不断提高，x25的做法显得越来越因噎废食了，所以其改良版本Frame Relay，就逐渐替代了x25，FR抛弃了x25中很多无谓的传输保障机制，而仅仅留下一些流控机制。相对于以太网的不负责任，FR起码在链路层面，实现了比较好的流控措施。

但是，不管是以太网，还是FR，都没有实现端到端的传输保障。端到端，是相对于“过路”来说的。过路是指在两个终端之间通信路径上的网络设备之间的路径。链路层的传输保障就是一种过路保障，因为链路层只保证相连的两个设备之间传送数据正常无误，但是不能保障通信最终端接收和发送的数据正常无误。因为在一个典型的包交换网络中，数据包一般都是一跳一跳地被传送的，每一跳两端的设备用链路层协议进行传输保障。

但是最终目的是要让通信的最终两端无误地收到数据，才能算作真正的传输保障，即端到端的保障。而FR协议所做的，只是在过路的时候保障链路正确传输。如果链路正确传输给了终端，而终端到最终上层的某个环节出错了，那么数据同样也是错误的，所以，要实现端到端的传输保障，一定要在最终传输终端上运行一个侦错和纠错逻辑，用来发现链路层所发现不了的错误。图13-1为端到端保障与过路保障的示意图。

图13-1 过路保障与端到端的保障运行中IP及其下层链路层协议

4. TCP保驾护航

为了实现这个目的，TCP出现了。TCP作为一个程序运行在通信的两个终点，不管两点之间用什么样的链路连接，经过了多少网络设备，TCP程序始终运行在通信终端上，监控终端最终发送和接收到的数据包的顺序、缓存区、校验等信息，检查是否出现丢包、阻塞等事件，一旦发现错误，立刻纠正重发数据包。

TCP不是运行在通信路径上的，而是运行在通信终点的两端设备上。即使过路链路保障机制再健全，TCP也是有必要的，因为数据包只有被终端正确接收到，才能算真正的传输保障。

所以，在IP之上，又凌驾了一层TCP逻辑，用来保障端到端的无误传输。而FR等链路层协议的保障机制，只能保障本段链路传输无误，不能保障端到端的正确收发，所以只能沦为数据链路层协议的角色了，用来承载IP和TCP。

我们可以体会到，协议之间也是在互相利用，互相排挤、吞并及融

合，以适应不同的应用环境，因为不可能为每一种应用环境都设计一种协议，协议之间互相利用、融合，才是最好的解决办法。

5. 最佳拍档——**TCP/IP**和以太网

现在可以回答上面没有找到答案的那个问题了，为什么以太网偏要和**TCP/IP**组合成一对呢？因为以太网使用得太广泛了，而OSI的第三层、第四层，也几乎被**IP**、**TCP**给统一了，所以以太网+**TCP/IP**，当然就成了一对好搭档了。

虽然一个协议可能实现OSI的所有7个层次，但是如果它要和其他协议合作，那么就要有个分工，而不能越权，比如**IPoA**、**ATM**只要传输**IP**包到目的就可以，而不管数据是否出错、乱序等，虽然**ATM**可能有这个功能。以太网虽然自己可以寻址，但是它还是配合**IP**，进行**IP**到**MAC**的映射，统一使用**IP**寻址，它默默无闻，所有光辉都被**TCP/IP**所披挂。

13.3 网络通信协议的四级结构

网络通信协议，一般可以分成Payload层、信息表示层、交互逻辑层和寻址层。其中最重要的是交互逻辑层，它是一个协议的灵魂。

1. Payload层

Payload是协议所承载的与本协议逻辑无关的最终数据，是通信终端通过本协议最终需要传送给对方的数据。Payload也就是协议所运输的货物。Payload层中的数据，既可以是最终应用产生的数据，也可以是另一种协议的信息表示层+Payload数据。如果Payload封装的是最终应用产生的数据，则表示这个协议是直接被上层应用程序来调用，从而完成程序之间的远程网络通信的。

如果Payload封装的是另一种协议的信息表示层+Payload数据，那么就证明这个协议此时正在承载那个协议。比如协议A封装了协议B的信息表示层+Payload，则可以说协议A封装了协议B，或者协议A承载了协议B，或者说协议B is over 协议A（BoA）。我们后面会描述一种协议被Map（映射）到另一种协议，而不是被封装，这种融合方式称为AmB，是彻底的协议转换，而不是仅仅做隧道封装。

2. 信息表示层

信息表示层就是附加在Payload数据之外的一段数据，也称作协议开销，因为这段数据和最终应用程序无关，是运行在通信双方的通信协议用来交互各自的状态，从而使双方做出正确动作的一段重要数据。这段数据可以想象成提货单或者信封。信封封装了信纸，信封上的地址、

姓名等信息，就是信息表示层，它可以让对方检测到当前通信所处的状态。

3. 交互逻辑层

这一层其实就是运行在通信双方协议系统上的动作程序代码逻辑，它根据对方传送过来的信息表示层数据来做出相应的动作逻辑，再生成自己的信息表示层发送给对方，然后对方再做相同的处理判断动作，就这样完成通信双方之间的正确动作。交互逻辑层其实就是协议的设计思想。交互逻辑层对于每种协议都不相同，但是很多都类似，可以说网络通信协议基本思想是类似的，因为它们所实现的目的都是一样的，就是将数据通过网络传输到目的地。

正因为如此，各种协议的交互逻辑层才可以相互融会贯通，将一种协议的逻辑，映射翻译到另一种协议的逻辑，从而将各种协议的优点结合起来，完成目标。协议逻辑层一般都是运行在通信双方两端的，但是像IP路由协议等，通信双方经过的路径上的所有设备，也都需要运行，因为IP包是一跳一跳被接收并且转发的。

4. 寻址层

它是帮助协议来找到需要通信的目标的一套编址和寻址机制。比如IP地址、MAC地址、DLCI地址、电话号码等。如果是点对点传输协议，则可以忽略此层，因为不需要寻址。而且不同协议之间的寻址层，可以互相映射翻译。

以上的这四层，是任何一个网络通信协议所必须具备的，不管多么简单或者多么复杂的协议。

5. 通信协议的相似性

相似性是通信协议之间相互融合的一个条件。而协议之间相互融合的另一个促成因素，就是协议使用广泛程度不同，有时如果要完成一个目标，不得不借用某种协议。

就像TCP/IP协议，TCP/IP协议占领了全球Internet的领地。如果有一种协议想跨越地域或国家来进行通信，但是自己又无能为力，因为它首先没有专门为它准备的物理线路，其次它的设计，也不适合大范围、长距离的广域网环境，那么它只能来租用TCP/IP协议，将自己封装到IP包中传送。能适合Internet规模的网络通信协议，唯TCP/IP莫属！而其他协议想要完成Internet范围的通信，就不得不借助TCP/IP，搭TCP/IP的车，让TCP/IP来承载它们。它们是怎么搭上TCP/IP的快车呢？

我们不妨类比一下。在整理本章的时候，恰逢大连刚刚开通了一艘新的火车箱滚装船。我想用这个例子来比喻协议融合，再适合不过了。从山东烟台到大连，最近的路径就是走渤海湾水路，如果搭乘陆路火车，则需要绕一大圈，所以很多货运汽车，甚至火车，都选择乘船到大连，下船后，车厢用火车头拉走，这样，在增加很少成本的条件下，节约了大量时间。协议融合同样遵循这个原则，只要能使总体拥有成本降低，性价比提高，任何协议都可以融合。

13.4 协议融合的三种方式

协议和协议之间的相互作用，有三种基本的思想。

- 第一种是调用（Use），也就是一种协议完全利用另一种协议。
- 第二种是隧道封装（Tunnel），一种协议将另一种协议的完整数据包全打包隧道封装到新协议数据包中。
- 第三种是映射（Map），也就是一种协议对另一种协议进行映射翻译，只将原来协议的Payload层数据提取出来，重新打包到新协议数据包中。

1. 调用关系

所谓调用，也就是一种协议自身没有某些功能，需要使用另一种协议提供的功能。比如TCP调用IP，因为TCP没有寻址功能，所以它利用IP来寻址。而IP又可以调用以太网，因为IP只有寻址功能，它没有链路传输的功能，所以它利用以太网提供的链路传输（交换机、Hub等）。IP调用PPP来传输等，也就是上层协议为了达到通信目的，使用另一种协议为其服务。这种关系严格来说，不算是融合。

2. 隧道关系

隧道封装，顾名思义，就是将一种协议的完整数据包（包括Payload和协议开销）作为另一种协议的Payload来进行封装，打包传输到目的地，然后解开外层协议的封装信息，露出内部被封装承载的协议完整数据包，再提交给内层协议处理逻辑模块进行处理。也就是说，进行协议转换的设备根本就不需要去理解内层协议到底是什么东西，到底

想要干什么，只要将数据包统统打包发出去。**Tunnel**的出现，往往是由于被**Tunnel**的协议虽然和外层协议都在某一方面具有相似甚至相同的功能，但是在某些特定的条件下，被**Tunnel**协议不比外层协议表现得优秀，不适合某种特定的环境，而这种环境，恰恰被外层协议所适合。这就像用船来装火车箱一样。**Tunnel**的另一个目的是伪装内层协议。

3. 映射关系

Map是比**Tunnel**更复杂、更彻底的协议融合方式。所谓**Map**，也就是映射，就是将内层协议的部分或者全部逻辑，映射翻译到外层协议对应的功能相似的逻辑上，而不是仅仅做简单的封装。**Map**相对于**Tunnel**，是内外层协议的一种最彻底的融合，它将两种协议的优点，融合得天衣无缝。内层协议的**Payload**层在**Map**动作中是不会改动的，因为**Payload**层的数据只有两端通信的应用程序才能理解。

13.5 Tunnel和Map融合方式各论

例如火车、汽车是两种运输工具，它们看似有太多的不同，但是它们的功能都是相同的，都是将货物运送到目的地。而火车需要在铁道上跑，但汽车需要在公路上跑（物理层不同，链路层不同）；火车因为铁轨很平滑，需要用钢铁轮子，而汽车因为公路很颠簸，需要用充气轮胎；火车不需要红绿灯来制约，而汽车在公路跑上，会有很多红绿灯来制约它；火车由于跑在专用的铁轨上，所以它能达到很高的时速，而汽车由于在共享的公路上跑，它能，但是不敢达到太高的时速；火车只能按照它的轨道来运行，而汽车几乎随处可去.....

以上列举出了火车和汽车的种种特点，相应地飞机、轮船、火箭等都可以拿来对比，这些特点就像各种通信协议自身的特点一样。同样都是运输货物，但是它们都适应了不同的需要。只不过网络通信协议运输的不是货物，而是一串0和1，是高低变化的电平，是数据，是信息。不同的通信协议同样也是为了满足不同的情况、不同的需求。TCP/IP协议满足了Internet范围的网络通信；FC协议满足了后端存储的专用高速公路这个环境，二者都各自占有自己的领地，谁也取代不了谁。就像铁路不可能为了和民航竞争，而把轨道往天上修，航空公司也不可能为了和陆运公司竞争，而让飞机跑在公路上。

TCP/IP适合整个Internet范围的通信，而SCSI协议不适合，所以如果SCSI协议需要跨越大范围通信，就要将其承载到TCP/IP上，也就形成了iSCSI协议，然而TCP/IP根本就不关心什么是SCSI，更不知道SCSI是怎样一种作用逻辑，它只是负责封装并传输。同样，因为以太网是个面向无连接的网络，没有握手过程，也没有必要有终端认证机制、没有NCP机制（PPP协议中用来协商上层协议参数的机制），而PPP却有这

些机制，它非常适合ISP用来对接入终端进行认证和管理，但是PPP的使用程度远远不如以太网广泛，怎么办？融合吧！于是形成了PPPoE协议。

13.5.1 Tunnel方式

iSCSI和PPPoE这两个协议，是典型的Tunnel模式。前面已经给Tunnel下过定义了。首先一种PoP的模式被定义为Tunnel的前提，就是这两种协议对某一特定的功能均有自己的实现。如果一种协议在某方面的功能，另一种协议没有实现，那么另一种协议就是“调用”那种协议，而不是被Tunnel到那种协议。比如，IPoE就是典型的调用，而不是Tunnel或者Map，因为IP没有链路层功能。

注意： IP与Ethernet之间的编址逻辑是映射关系而不是使用关系，即IP地址与MAC地址的相互映射。

用iSCSI来分析，TCP/IP可以实现寻址和传输保障，SCSI协议也可以实现寻址和传输保障，所以它们具备了这个前提；同样，PPPoE也是一种Tunnel方式的融合协议，因为PPP和Ethernet都是链路层协议。

1. VPN的引入

Tunnel的另一个作用，就是伪装。有时候虽然两种协议实现的功能、适用环境都相同，但还是将其中一种Tunnel到另一种之上，这是为什么呢？有些情况确实需要这种实现方式。比如IP协议中的GRE，通用路由封装，就是这样一种协议。它将IP协议承载到IP协议本身之上，自己承载自己，再封装一层，这样就可以使得一些不能在公网路由的IP包，封装到可以在公网路由的IP包之中，到达目的地后再解开封装，露

出原来的IP包，再次路由。这就是伪装。

利用这种思想，人们设计出了VPN，即Virtual Private Network，用来将相隔千里的两个内部网络，通过Internet连接起来，两端就像在一个内网一样，经过Internet的时候，使用公网地址封装内网的IP包。这是最简单的VPN。在这基础上，又可以对IP包进行加密、反修改等，形成IPSec体系，将其和原始的VPN结合，形成了带加密和反修改的IPSec VPN，真正使得这种PoP穿越外层协议的时候，能够保障数据安全。

2. 例解Tunnel

下面再举个例子来说明，到底什么是Tunnel。

邮政系统，目前已经是举步维艰。21世纪之前，网络还不很普及，除了电话、电报，写信似乎是大家长距离通信的唯一选择。寄信人将自己的信件（数据，Payload）装入信封（协议信息表示层数据段），填好收信人地址、邮编、名称（通信协议的信息表示层、寻址层）等，交给邮局（网络交换路由设备），由邮局进行层层路由转发，最终到达目的地。

IP网络和邮政系统极其相似。而为什么邮政系统目前已经陷入了困境呢？原因就是竞争。

进入21世纪之后，物流业快速兴起，它们借助公路、水路、航路、铁路等“链路层”，加上自己的一套流程体系（协议交互逻辑），充分利用这些资源达到物流目的。以前只有邮政一种方式，而现在出现了许多的物流公司，每个公司都有自己不同的物流体系，但是基本思想大同小异，都是要将用户的货物运送到目的地。

21世纪，虽然网络已经很发达，但是网络只能走信息流，走不了实物流。所以物流公司还是能占据一定市场。

提示： 我们来看看21世纪，用户是怎么来寄出一封信件或者包裹的。同样寄出一封信，如果还是用古老的协议，比如信封+80分邮票的形式，还是可以的，大街上现在还有邮筒。但是很多快递公司也提供信件包裹服务，只不过他们用的信封，比普通信封大、结实，而且他们信封上的标签，所包含的信息更加具体和丰富，比如增加了收件人电话、发件日期、受理人签字、委托人签字等。邮政信封具有的，快递信封都具有。

这样就可以看出这两种协议的不同之处了。用户可以把信件封装到邮政普通信封直接发送，也可以封装到快递公司信封中发送，也就是选用其中一种协议。

那么如果用户先把信件（最终数据）封装到普通信封中，填好信封头信息（协议信息表示层和寻址层），然后将封装好的普通信封，再封装到快递公司的信封中，并再次填一份快递公司的信封头信息；快递公司按照这些信息，将信件送到目的地，目的收到之后，解开外层信封，然后解读内层信封的信息头，再次转发，或者直接打开。刚才描述的这种情况，就是一个典型的协议Tunnel方式的相互作用，把邮政协议Tunnel到快递公司的协议，这种Tunnel的目的，就是为了获得快速、优质的服务，因为普通邮政协议提供不了快速高效的服务。

思考： 我们再来看这种情况，比如快递公司A，在北京没有自己的送货机构，但是青岛有人需要向北京送货，怎么办？

此时当然要考虑借助在北京有送货机构的快递公司B，让他们代

送，将信件封装到快递公司A的信封，然后再将A的信封装入快递公司B的信封，让快递公司B做转发，到目的地之后，B的送货员剥开外层信封，最终用户会收到一个快递公司A的信封，客户就认为是快递公司A全程护送过来的，其实不是。这样就很好地伪装了信件。这是Tunnel的另一个目的。

13.5.2 Map方式

说完了Tunnel，我们再来说Map。Map就是将一种协议的逻辑，翻译映射成另一种协议的逻辑，Payload数据完全不变，达到两种协议部分或者完全融合。

还是快递公司的例子。两个快递公司（两种协议），快递公司A在青岛没有自己的送货机构，但是B有。所以A和B达成协议，A将青岛地区的送货外包给B，凡是A公司在青岛的业务，都由B来运送，但是表面上必须保持A的原样，这种方式目前实际已经广泛使用。起初的做法是：先将客户信件装入A信封，然后再封装一层B信封，带着A信封来转发，也就是Tunnel。后来，B公司嫌这种方法浪费成本，因为额外携带了一个A信封，这增加了信件的重量和信封成本。所以B公司琢磨出一套方法：

先让B公司的取件人了解寄件人所要提供的信息，此时取件人担当A公司的角色，用户认为取件人是A公司的，用户按照A公司的协议，将信封头信息告诉取件人；然后取件人此时并没有将信件装入A公司信封，而是直接装入了B公司信封，但是在填写B公司信封头的时候，取件人将用户提供的针对A公司特有的信封头信息，转换翻译成B公司特有的信封头信息；经过B公司转发，到达目的地之后，送货员再次将B公司的信封头信息，转换翻译成A公司所特有的信封头信息。

这样，两端的用户，同样也丝毫不感觉不出中间环节其实是B公司完成的。但是这种方式相对于Tunnel方式的确节约了B公司的成本，使得开销变小了，提高了转发效率。这种方式的协议之间的相互作用，就是Map。

1. IP和以太网之间的寻址关系Map

最简单的Map就是IP和以太网之间的寻址关系Map。IP地址必须映射到MAC地址，才能享受以太网的服务。正如IP和以太网之间的Use+Map关系一样，实际上，各种协议之间的相互作用，不可能只是其中一种作用方式：寻址体系之间一定需要Map（同种协议自身Tunnel的情况除外），交互逻辑层可以Tunnel，也可以Map，Payload一定需要Tunnel。所以针对协议不同的层次，都有相对应的相互作用方式。

2. 协议交互逻辑的Map

协议交互逻辑的Map，比寻址层的Map要复杂得多。寻址层的Map只要维护一张映射表就可以，交互逻辑的Map则需要维护一个代码转换逻辑模块。

两种协议的状态机的互相融合作用是很复杂的。比如TCP的流控机制和FC协议的流控机制之间的Map，TCP是靠窗口机制实现端到端的流控，FC靠Buffer to Buffer（过路流控）和End to End（端到端流控）两种机制实现流控。如果把FC协议承载到TCP/IP协议之上，那么就会出现Tunnel模式和Map模式，当然Tunnel中也可能需要Map，Map中也同样需要一定的Tunnel成分。

我们不妨称作：以Tunnel为主的模式和以Map为主的模式。

如果是Tunnel为主的模式，那么TCP/IP根本不管FC协议的交互逻辑是怎样的，TCP仅仅把FC当成Payload来封装并传送。

而Map模式中，进行Map操作的设备或者软件，就需要既了解TCP/IP协议的交互逻辑，又了解FC协议的交互逻辑，因为只有了解了双方的逻辑，才有可能进行Map。比如，FC协议发出了一个信号，说本方缓存将满，请降低发送速度。Map设备收到这个信号之后，就会Map成TCP/IP可识别的信号，即本方处理受阻，窗口减小至某某数值，这就是FC协议到TCP/IP协议关于流控机制Map的一个方法。

如果在Tunnel模式中，FC协议发出的这个流控信号，则会被TCP/IP给Tunnel传送到对方，然后再由对方的FC协议模块来根据这个信号来判断流控机制应该做出的动作，动态调整发送速率。

注意： 这个信号是直接原封不动地被传送到FC协议的对端处理机上处理，而不是像Map模式中在本地就终结了FC逻辑。

Tunnel模式中，TCP/IP不参与任何FC协议内部的逻辑。

除了FC流控逻辑的映射，其他Flogin登录机制、连接机制等映射，也都有自己的实现。比如，FC发起一个Plogin过程，那么Map设备可以Map到TCP/IP的一个握手过程等。

提示： Tunnel和Map这两种模式，在第8章还有一个将FC AL的环接入FC Fabric中的例子。

13.6 FC与IP协议之间的融合

哗啦……，早晨的微风把IP吹醒。原来IP做了一场美梦。根据梦中的指示，IP鬼使神差地将FC协议映射到了IP上。并做了两种模式，一种是以Tunnel为主的模式，称做FCIP；另一种是以Map为主的模式，称做IFCP。

在FCIP模式中，通信的双方各增加一个FCIP网关，任何FC协议的逻辑，哪怕是一个小小的ACK帧，都需要封装到TCP/IP协议中传输。两端的FC协议处理机不会感知到中间TCP/IP的存在，它们认为对方就是一个纯粹的FC设备。

在IFCP模式中，通信的双方各增加一个IFCP网关，作为协议转换设备使用。IFCP GW将FC协议终止在本地，提取Payload数据，对外以TCP/IP设备的形式出现并传输数据，到达对方之后，对方的IFCP GW再从IP包中提取出Payload，然后将其封装到FC帧中，对其内部以FC设备的形式出现。通信双方中间的TCP/IP协议，将大部分或者全部FC的逻辑都映射成TCP/IP的逻辑。

比如每当一个FC设备需要和远端的FC设备通信，发起Plogin，那么IFCP GW就向对方建立一条TCP连接，用多条TCP连接和不同的IP地址来区分不同的FC设备。此外，还需要保存一个TCP端口或者IP地址对FC设备24b的Fabric地址的映射表。如果两端的FC设备的ID有冲突，这个映射表还需要考虑NAT，将地址翻译成其他ID。相对于IFCP，FCIP协议则不能识别FC的逻辑，因为它只是Tunnel，如果两端Fabric中有ID冲突的，那么也只能冲突着了。

至此，FC协议终于可以享受TCP/IP带来的扩展性了，FC搭上了TCP/IP的车，远隔千里都可以跑上FC协议了。IP 大获成功！IP 和FC从此握手言和！

伟大的**SCSI**协议

可以说整个网络存储系统，都起源于一个协议体系，这个协议体系就是SCSI协议。网络存储的任何内容，最终都是为了将这个协议体系发扬光大。人们将这个协议强行划分解体成了多个层次，然后把它的最上面的几层，与另一个协议体系——Fibre Channel协议的下几层进行融合，形成了FCP协议，这种协议目前运行在各个厂家的高端磁盘阵列上。曾经一度时间，以太网甚至也看好了SCSI协议，想与其融合成所谓的“ESCSI”协议，但结果没有成功。以太网失败之后，它的好兄弟IP接着跟上，最终成功地与SCSI协议进行了融合，生成了iSCSI协议，目前也被广泛应用于一些低端盘阵。

为何不是ATA或者FATA呢？原因就是SCSI协议体系本身就比ATA协议体系高效并且功能强大，此外，SCSI的硬盘性能也普遍比ATA硬盘转速快，性能高，用于服务器系统，所以SCSI当然是首选了。另外，一个巴掌拍不响，SCSI协议本身就想把自己给“嫁”出去，因为它很早就已经迫不及待地将自己分成了很多层次，来吸引其他协议。

协议融合的结果，就形成了目前形形色色的网络存储世界，各种融合协议、各种产品、各种解决方案，好不热闹！而原本的SCSI协议，除了一些磁带机以及主机本机硬盘外，已经不再使用。SCSI融合入了各种协议中，它无处不在，虽然它的躯体已经是七零八落，但是它的精深思想，以及为技术而献身的精神，将在形形色色的技术中永放光芒！

13.7 无处不在的协议融合

之所以提出“协议融合”这个名词，而不是“协议映射”或者“协议隧道”，是因为“融合”这个词更加通俗易懂；另外，也更加具有生物学色彩。计算机就是人类所创造的另一种形式的“生物”，人类就是计算机的上帝。

1. 协议融合和基因融合

分子生物学家们将不同功能的基因段整合到一起，再用核糖体蛋白质机器读取其代码，表达成肽链，然后折叠成三维结构的新功能蛋白质分子，比如抗冻小麦、发光的白鼠等。这就是基因融合。这个过程与协议融合类似。

协议融合是无处不在的，正如不同快递公司之间的合作一样。甚至连劳动合同方面都出现了融合，劳务派遣公司与劳动者签订合同，然后将劳动者输送到用工单位工作，用工单位不必维护人事系统，将人事系统外包给劳务派遣公司。

2. 航空公司的协议融合

目前，国际上大多数的大型航空公司都利用IBM或者Unisys的大型机系统作为订票和离港系统的处理机。世界各地的售票和离港终端都通过某种网络系统与大型机连接并且通信。航空业的大型机与终端通信协议也经历了纯种和融合阶段。

IBM利用ALC协议与其终端通信，Unisys主机则通过UTS协议与其

终端通信。但是随着IP网络的成本不断降低，质量不断提高，UTS和ALC这两种古老的纯种协议，不得不考虑将自己嫁给IP网络，从而出现了MATIP协议，也就是将这些协议承载于IP之上。Cisco公司也为航空业专门开发了这种融合协议，称为ALPS协议。然而ALPS最终没有成为RFC标准，而MATIP协议，却最终登上了RFC宝座。MATIP协议的文本可以查看RFC2153。

13.8 交叉融合

提示： 在本书写作之时，FCoE这个由FCP与以太网结婚所产生的融合协议，正在被一些厂商炒作得沸沸扬扬。FC协议与SCSI协议融合之后形成FCP协议，而FCP协议又与Ethernet融合形成FCoE协议。

如图13-2所示为协议融合树。

从图13-2可以看到，FCP协议与IP协议融合的后代是双胞胎。各种协议之间相互融合，甚至产生了交叉，但是一切融合都是为了更好地适合市场需求。

图13-2 协议融合树

提示： 另外，由于IP网络的大肆普及，众多的协议动辄就要非IP不嫁，而IP和以太网绝对是铁哥们儿，所以以太网也借IP的光，就凭着自己仅仅1Gb/s的带宽到处招摇撞骗，这不，成功地把比它多3Gb/s带宽的FCP协议给忽悠过来了。不过以太网也在潜心修炼，等练成出关之后，其10Gb/s的速率，将会让人望而生畏，但愿那时候以太网统一天下！

13.9 IFCP和FCIP的具体实现

上面说到IP根据网络通信协议之间的相互作用，成功地将FC协议和TCP/IP协议进行了融合，生成了一种FCIP的Tunnel协议和一种IFCP的Map协议。

蓝图有了，那么具体怎么来将其实现呢？我们知道，不管是FCIP的简单Tunnel模式，还是IFCP的复杂Map模式，进行这种PoP操作的角色，一定是一端面对FC协议的网络，另一端面对TCP/IP协议的网络。

1. 协议转换器

同时面对多种协议，并在多种协议之间实现相互融合、相互转换的设备，就称做协议转换器。如果这个转换器只是起到一个桥联的作用，只在一条链路上串联，那么就称其为协议桥接器。如果这种转换器，不但要实现单条链路上的协议转换工作，而且还需要实现一些转发动作，即在多条链路、多个网络之间互相转发数据，则可以称其为协议路由转换器。如果某种协议路由器可以实现多于两种协议的网络互联，则称其为多协议路由转换器，因为它能在多种协议之间互相转换并做路由转发。

SAN要想获得扩展性，即要想将相隔两地很远的两个SAN网络通过IFCP或者FCIP连接起来，就必须在双方的SAN系统前端各增加一个协议转换设备，这个设备后端连接各自的SAN，前端连接IP网络，在广域网络上运行FCIP或者IFCP协议通信，达到协议转换的目的。

两个独立的系统连接起来，就涉及了两种情况。

- 第一种：两个系统连接之后，在逻辑上还是独立的，即一个系统不影响另一个系统，但是它们之间可以通过协议转换设备来通信。
- 第二种：两个系统融合成一个大的系统，逻辑上是一体的，只不过相处两地，之间用协议转换设备连接。就像以太网一样，如果用光缆将两地的两个局域网直接连接起来，两地的系统同在一个广播域中，这样就相当于把两个系统融合起来了。

但是如果两地各自接一个IP路由器，广域网链路上承载的是基于广域网链路协议之上的IP包，那么两地的局域网就没有被融合，只是可以相互通信而已。

提示： 有的时候，两地的系统必须融合，而有的时候，不需要融合。是否融合，需要看最终的需求。所以协议转换设备也必须能够处理这两种情况。对于需要融合的情况，协议转换设备不需对两端的SAN逻辑做任何附加处理，而只需要将两端的逻辑Tunnel或者Map到广域网协议上就可以了；而对于不需要融合的情况，协议转换设备就需要对两端系统的逻辑做一系列的处理、屏蔽、虚拟和欺骗了。

2. TCP/IP和以太网网络实例解析

我们不妨拿TCP/IP和以太网网络来做一个例子。

假如一个公司，在a地和b地，分别有一个办事处，每个办事处有一台以太网交换机，上面各连接了几台终端。现在为了业务资源共享，公司决定将两地的网络融合起来。公司向ISP申请了一条2Mb/s的E1专线（当然也可以申请Internet线路，两端都接入Internet，然后做L2VPN或

者L3VPN）。

公司有两种选择方案。

一种是直接用这条专线把两地的交换机连接起来，在这条线路上直接承载以太网帧。

另一种选择就是两端各加一个路由器，隔离两边的局域网，但是保持它们之间的通信。

这个公司最终选择了后一种方案，原因就是保持了双方的独立性，同时保证性能。因为毕竟是两个办事处，如果彻底进行融合，不但不安全，也不利于扩展，而且容易造成广域网流量太大，因为彻底融合之后，以太网广播就要跨广域网来互相传递，这无疑是浪费资源的。在隔离的基础上，同样能够保持双方无障碍的相互通信，只是不能像在一个局域网内那样直接利用MAC来点对点通信。如果a地某个节点需要和b地某个节点通信，a地的这个节点需要先把数据发给a地的路由器，也就是网关设备，然后让网关来转发给b地。虽然增加了一层操作，但是这样做的可扩展性、可管理性都增强了。在路由器上可以做访问控制、地址转换、QOS、策略路由等基于IP甚至TCP层次的个性化动作。如果是直接局域网融合，则这些特性都不能实现。

3. SAN系统实例解析

再来看SAN的情况。还是这个公司，a地和b地各有一个SAN系统。为了实现存储资源直接共享，公司决定将这两个SAN联通起来。同样也存在两种情况，即彻底融合或者相对独立的连通。

如果是彻底融合的话，那么广域网链路就完全相当于一条ISL链

路，只不过通信协议可能是FCIP或者IFCP协议。

对于FCIP，任何FC帧都将被透明地传递。对于IFCP，一部分FC帧会被屏蔽或者MAP。但是这些被屏蔽或者MAP的帧，都是和底层通信有关的，而上层逻辑性质的帧，IFCP也需要透传到对端。

这些业务逻辑性质的帧，比如RSCN帧，用来传递Fabric网络中的重要变化信息给已经注册了这项服务的节点；再比如Plogin、Process Login等这些都是业务逻辑性质的，和底层通信无关。

彻底融合之后，两个SAN系统就融合为了一个系统，那么这个系统就会有一个主交换机。主交换机为系统中其他交换机分配域ID，并且两个交换机之间需要运行FSPF路由协议，不停地发送一些路由控制帧，再加上主交换机选举时产生的帧，主交换机失败时，整个Fabric的重建过程中每个交换机发出的各种帧都需要经过广域网链路进行传送。

不但这些帧要占用广域网带宽，而且一旦主交换机发生故障，那么对方的SAN系统会进行Rebuild，包括重新选举主交换机、重新建立路由表等，这个过程中，IO就会暂时中断。

注意： 由于广域网链路速度相对慢，稳定性相对差，所以一旦这条链路发生不稳定的振荡，那么就会造成主交换机重新选举。如果链路频繁闪断的话，那么两端的SAN系统根本无法正常工作。

所以说，两地SAN系统彻底融合的话，一旦某地的系统故障，就会影响到另一个系统的正常运行，而且要占用额外多的宝贵的广域网资源。由于访问存储资源对性能和延迟要求较高，所以彻底融合两个SAN，最好只在局域网内进行，交换机间的链路最好是裸光缆或者高速

链路，否则最好采用另外一种融合方式，即逻辑独立、全局连通的融合方式。

13.10 局部隔离/全局共享的存储网络

将SAN系统彻底融合，扩展性差、管理性差，而且耗费广域网链路资源。所以这个公司同样也选择了相对独立的连通方式。下面来看一下相对独立的融合到底是个什么概念，它的作用机制是怎样的。

“a地的SAN交换机（E端口）—a地协议转换器—广域网链路—b地协议转换器—（E端口）b地SAN交换机”这种拓扑不管是彻底融合，还是独立融合都一样，只不过协议转换器在两种方式下所做的工作不一样。彻底融合方案中，协议转换指Tunnel或者Map通信底层的协议逻辑，而不管上层业务逻辑，也就是只要从E端口收到了帧，协转就将其Tunnel或者Map到IP协议中发送给对端。而相对独立的融合，不但要Tunnel或者Map底层协议逻辑帧，它还要理解FC的上层逻辑，做到“报喜不报忧”。

独立融合/全局共享

所谓独立融合，就是说两端的SAN系统都可以独立运作，而不依靠另一方，或者受另一方的影响。这样就不能像彻底融合那样一端为主交换机，一端为非主交换机，而要让两端独立起来。由于两端的Fabric中都各自只有一台SAN交换机，所以两端的SAN交换机都是主交换机，各自为政。

既然这样，怎样和对方的SAN进行通信呢？协议路由器自有其招数。协议路由器与SAN交换机之间通过E端口连接，它欺骗两端SAN交换机，让交换机认为它正在连接着另一台交换机，而这个由协议转换器虚拟出来的交换机级别比它低，所以它自己认为自己就是主交换机。虚

拟交换机和SAN交换机之间运行FSPF路由协议，所以这个虚拟交换机就获得了SAN交换机下面所有连接的终端节点信息。

获得这些信息之后，a地的协转通过广域网链路将这些信息通告给b地的协议转换器。b地的协议转换器同样和b地的SAN交换机之间运行着FSPF路由协议，同样也欺骗了b地交换机。b地协议转换器收到了a地协议转换器发来的关于a地SAN交换机上所连接的所有节点信息之后，就利用和b的SAN交换机之间的FSPF路由协议，将这些节点信息通告给b地SAN交换机，所以b交换机就有了a交换机上节点的信息，同样a交换机也会拥有b交换机上节点的信息，这样，a和b交换机之间就可以通信了，其实它们都不知道中途有两个中介在骗它们。

如果其中一个SAN系统发生故障，那么这个系统中的协转设备，会将这个重大消息屏蔽，不告诉对端的SAN系统。因为一旦被对方系统得知，便会发生Fabric的重建过程，影响本端SAN系统的IO。有了SAN路由器，远端SAN访问的超时，并不会影响本地SAN的访问。此即所谓“报喜不报忧”。同样，一个SAN系统中的诸如RSCN等广播类的帧，也会被协转设备根据策略而终结在本地，不会跨越广域网链路通告给对方。协转设备还应该具有访问控制功能。

这种方案被称做“SAN路由”，因为它具有与IP路由类似的功能和架构。

13.11 多协议混杂的存储网络

多协议混杂的存储网络如图13-3所示，其中的中枢引擎是两个互相连接的多协议路由器。这个多协议处理机，就像一台计算机的CPU，Fabric和以太网就像计算机的IO总线，磁盘便是计算机的外设和输入设备，各种存储控制器便可以理解为计算机上的各种IO控制器，前端的Fabric和以太网便是前端的IO总线，主机服务器则是输出设备。即磁盘上的数据，经过输入总线输入CPU进行运算，然后通过输出总线，输出给主机服务器。这又是一个轮回，不折不扣的轮回，循环嵌套，永无止境。

图13-3 多协议混杂的存储网络

图13-3所示的拓扑，可以说是一个大的统一的拓扑。存储网络不外乎就是图13-3中列出的元素。磁盘经过一层层的IN和OUT，一层层的虚拟化或者桥接透传，最终被主机看作是一个LUN或者目录。不妨将其抽象，隐去复杂的部分，就形成了图13-4的拓扑。

图13-4 一次抽象后的系统架构

再抽象一下，得到如图13-5所示的模型。

图13-5 本质模型

13.12 IP Over FC

FC可以为了提高扩展性而与IP合作，同样，IP也可以Over到FC上来搭个快车过把瘾。难道可以给每个FC口设置一个IP地址甚至多个IP地址，然后上层的应用程序就像使用一块网卡一样使用FC卡么？当然可以，FC本来就是一块网卡，只需要在链路层之上封装一层PoP逻辑即可。比如，给FC卡上的每个接口配置一个IP地址和掩码，甚至网关，然后对目标IP地址发起Ping操作，OS接收到Ping操作指令后，随即调用到ICMP协议层，然后下到TCP/IP层，TCP/IP层的下位就是FC卡的驱动程序层了，PoP逻辑正是生效于此层。这个特殊的驱动程序首先检查目标IP地址，并查找它所维护的IP-WWPN地址映射表（这张表可以手动编辑，也可以使用ARP协议来自动发现和维护，基于FC网络的ARP协议在IETF协议规范中可以找到），找出这个IP地址在网络中对应的FC网络地址；找到之后，如果尚未发起Plogin过程则发起，否则直接将ICMP数据包打包入FC帧传送到对应的FC Target。FC Target收到数据包之后便依次向上层传递，当传递到驱动程序时，驱动程序解析数据包中的高层协议并向对应的上层调用程序，即TCP/IP发送，TCP/IP层处理后将其转发到调用应用程序层，即ICMP协议层。ICMP协议处理之后会做出相应反应，比如生成一个Echo Reply数据包，发送到请求端，发送过程与之前类似。

哟.....又吹，做梦呢吧？我清醒着呢，虽然现在快午夜12点了，但是我依然保持文如泉涌。Qlogic公司早已发布了这种IP Over FC的驱动程序。目前，其QLE2464型号的4端口4GB/s的FC适配器（如图13-6所示）在Windows系统中已经可以在这款驱动程序下实现IP Over FC功能，安装这个驱动之后，系统会在网络适配器列表中生成一个特殊的网

络设备，其使用与普通以太网卡类似，而且可以对其配置IP地址等参数。

图13-6 QLE2464适配卡

有了这样的设计，就可以在FC的链路层上透明地使用iSCSI或者NAS协议来访问数据了。这样做的目的只有一个：速度！让iSCSI和NFS/CIFS也可以享用FC带来的4GB/s甚至8GB/s的速度，搭上快车兜风的感觉，爽！

话锋回转。FC为了O到IP上，整出了一对双胞胎协议。然而，IFCP或者FCIP协议都需要一个外部硬件协议转换器来实现，为何不能像IPoFC一样搞法呢？FC虽然也可以按照OSI模型分层，但是其链路层和网络层甚至更高层一般都是运行在FC卡上的，要实现PoP逻辑，被承载协议逻辑必须运行在承载协议逻辑的上层。所以要将FC承载于IP之上，就要求FC的网络层及以上的所有层次必须作为软件运行在主机上而不是FC卡上，然后主机再将封装映射好的数据包通过TCP/IP及以太网卡发送出去。目前，有一些低端的FC卡的网络层及以上逻辑确实运行在主机上，也就是运行于FC卡的驱动程序中，而FC适配卡上的芯片只负责实现FC的物理层和链路层功能（这种低端软FC卡其需要利用主机CPU来实现FC的上层逻辑，所以CPU占用率会比纯硬FC卡，在主机CPU负载比较高的时候，就会影响底层驱动程序的效率造成IO性能受到一定影响）。如果将FC卡变为以太网卡，而保留原有FC卡驱动中的FC上层逻辑处理层，即FC网络层及以上的处理逻辑，然后在以太网卡和FC处理逻辑之间插入TCP/IP协议栈，之上再插入一层PoP逻辑，PoP逻辑生成虚拟的WWPN地址来欺骗上位的FC处理层，利用ARP协议来获取和维护网络中的地址映射，这样做理论上可以达到单纯的FCoIP的效果。但是回头想一想，这是没有任何意义的！

仔细想一下，FC为何要O到IP上呢？就是因为IP可扩展能力强，那么如今我们在一个没有FC任何硬件参与的情况下，强行在TCP/IP协议之上增加了一层额外的与TCP/IP功能类似的网络协议，然后再在二者之间插入映射翻译层，这不是没事找事么？没有FC硬件参与，以太网网络硬件之上的TCP/IP之上的软FC上层逻辑，这简直就是一个累赘。我们直接用iSCSI或者NFS/CIFS不就得了么？干嘛非要在其上安插一个累赘呢？

同为FCoIP的实现方式，那为何IFCP和FCIP就不是累赘呢？因为IFCP和FCIP是为了将一个现存的、由众多FC适配器和FC网络交换设备组成的Fabric网络与另一个Fabric相互连通而连通用的网络传输协议为IP。所以，如果想让两个以太网之间互相连接而互联通道使用高速的FC连接，这便是有意义的。

哦，那么刚才说的IPoFC，以此类推是不是也可以被定义为没有任何意义了呢？这就需要从另一个角度看了。因为TCP/IP已经普遍使用，基于TCP/IP的存储协议也很多，比如iSCSI、NFS、CIFS、FTP甚至HTTP等，而基于FC的存储协议呢？只有FCP。而为了让这些开放存储协议运行在比以太网更加高速的网络上，将它们O到FC上当然是有意义的。有人又说了，哦.....那么说如果FC也像TCP/IP一样有很多基于它的开放协议被广泛使用，那么很有可能也有人开发单纯FCoIP了么？这又得从另外一个角度考虑，即使FC哪天达到像TCP/IP一样了，那么将高速的FC链路变为当前的1Gbps的以太网链路，就为了单纯的FCoIP，你干么？除非给它O到10Gbps的以太网上，这也就是FCoE的做法了。

图13-7和图13-8示出了FCoIP和IPoFC的区别。

图13-7 IPoFC示意图 图13-8 FCoIP示意图

13.13 FCoE

13.13.1 FCoE的由来

以太网的速率从1Gb/s直接跳跃到10Gb/s，今后还会再跳到40Gb/s、100Gb/s。FC呢，从1Gb/s跳到2Gb/s再跳到4Gb/s和8Gb/s，将来的规划还是乘2，也就是下一步是16Gb/s。FC总是想搭顺风车，前几年以太网只普及到1Gb/s，它看好了IP的扩展性，结果O了IP（IFCP和FCIP双胞胎）。结果这俩后代不争气，用的很少。O完了IP不过瘾，看到以太网迅速强大，自己却步履蹒跚，打算直接O到Ethernet上苟延残喘，也就有了FCoE。如图13-9所示，传统FC底层速率的发展已经明显落后于以太网了。

图13-9 FCoE、FC的路标对比

13.13.2 FcoE的设计框架

然而，并不像你所想象的那样，比如，iSCSI直接就是在主机端操作系统中的TCP协议栈之上加一层逻辑，直接利用现有的IP网络，那么FCoE是不是也类似，直接在主机端安装一个FCoE协议栈，直接将其承载到以太网上呢？如果真的是这样，FCoE与iSCSI就没有什么本质上的大区别了，也就无法体现其价值所在了。那么FCoE到底是一种什么形态呢？我们知道，传统FC环境下在主机端原本是靠一块FC HBA来实现FC网络接入的，全部FC逻辑都运行在这块卡上（SCSI逻辑运行在主机端OS内核中），这样做可以充分降低主机资源消耗。如果现在要将FC的上层逻辑全部拿来主机端协议栈中运行，那么效率势必要打折扣，对主机CPU资源的消耗也必将增加。那么iSCSI就不消耗主机资源么？当

然消耗，那为何iSCSI可以存在，就不允许FC的软协议栈存在呢？要知道，1Gbps的以太网环境下，iSCSI软协议栈不会消耗主机端太多资源（根据实测一颗单核Intel至强CPU约可承载300MB/s的大块连续IO的iSCSI流量，此时CPU已经满载，如果是IO密集型流量，CPU会更快满载），但是在10Gbps以太网的环境下，iSCSI软协议栈对主机CPU的消耗就不可同日而语了，有数据表明，10Gbps下的软iSCSI initiator对主机CPU的占用率在30%~60%之间，这已经不可容忍了。所以，10Gb/s以太网承载的iSCSI协议，迫切需要运行在硬iSCSI HBA上，这才是可以向企业级用户所交付的合适形态。至于前期微软与Intel合作进行的iSCSI百万IOPS测试结果（见之前章节），只是一个测试而已，恐怕其CPU利用率已经达到很高的程度了，这种测试不具备实用性价值，况且Intel是做CPU的，这种测试的目的大家都明白。所以同样，FCoE也需要运行在一块硬的HBA上。

13.13.3 FcoE卡

好，明白了这一点，我们再往下走一层。我们知道传统以太网并不提供链路层保障功能，而FC的传输保障在链路层与传输层都有实现，两者是紧密结合的。如果要将FC over到Ethernet上，那么势必要砍掉FC的链路层而用以太网链路层替代，但是这样over之后，新协议就没有链路层保障机制了，不能与FC传输层保障机制进行互相配合。为此，必须给以太网增加链路层传输保障机制，所以新的标准出现了，这就是CEE（Converged Enhanced Ethernet），即增强型以太网协议，Cisco称之为DCE（Data Center Ethernet）。这样，FC就可以没有后顾之忧地使用以太网作为传输链路了。

我们再往下走一层。以太网亲自出马改变自身来迎合FC的大驾，

这个举动可谓是兴师动众了，由于已经对链路层进行了改变，所以传统的以太网卡不可能符合要求，以太网交换机也是一样。这样的话，就得开发专门的以太网卡和以太网交换机来支持CEE了，确实是的。我们先来看一下以太网卡，前面说过，FCoE必须做成一个硬件HBA，而现在以太网卡也需要重新做，所以，一块类似iSCSI硬HBA的硬FCoE卡也就水到渠成地出炉了：CNA卡，Converged Network Adapter，将FC卡与增强型以太网卡结合的产物，FC卡把其上层的FC逻辑处理部分拿出来融合到增强型以太网卡中，然后再在这片卡件中增加一个FCoE协议处理模块，在数据帧中增加一个FCoE协议表示层区段（这个概念见本章之前部分）用以执行协议转换控制任务，所有的逻辑都集成到一块ASIC芯片中以达到很高的执行效率。图13-10所示为Brocade生产的某型号的CNA/FCoE卡的实物图，出口为两个10Gbps的以太光口。

图13-10 CNA/FCoE卡实物图

iSCSI硬HBA只能用作承载iSCSI协议，而不能作为承载IP的以太网卡使用了。但是FCoE卡却可以同时承载FCoE流量和承载IP的以太网流量，也就是说可以把一块FCoE卡同时当做FC卡与以太网卡使用，互不影响。

FCoE卡相比于FC卡来讲，除了在操作系统驱动层面有所变化之外，上层的协议栈比如SCSI层之类并没有大的变化。

13.13.4 FCoE交换机

好，我们接着往下走，再来看看以太网交换机的改变。除了在交换机端对应地增加对CEE的支持之外，以太网交换机还需要做一项重大的改变。我们知道FC交换机是整个Fabric的核心，任何一个通过FC接入

Fabric的节点都要首先获得一个Fabric ID然后执行Fabric Login过程，这些过程都需要FC交换机来参与执行。而以太网交换机并没有提供任何这些Fabric服务，这样的话，通过以太网链路传入的地址请求以及Fabric Login的帧，就没有人来应答和处理了。所以，以太网交换机上，必须增加对应的Fabric模拟程序，将其做到ASIC芯片中。

看到这里也许你已经完全糊涂了，不就是要利用起以太网么？iSCSI对以太网没有任何改变，而为何FC一来，就要兴师动众，改完了协议改网卡，改完了网卡还得改交换机，这么做到底值不值得？请各位带着这个问题继续阅读。

此时别忘了另外一个问题：原有的传统FC设备，如何与FCoE融合？既然叫做FC over Ethernet，那么就得出点样子看看，得将原有的传统FC网络融合进来才是。图13-11所示为一台Brocade 8000 FCoE交换机的逻辑架构图与实物图。

图13-11 FCoE交换机逻辑图与实物图

大家可以看到，这台交换机同时包含传统的FC交换模块及接口和新的CEE增强型以太网接口，这就意味着，传统的FC设备以及使用CNA卡的FCoE设备可以同时连接到这台交换机上而且实现互相通信。交换机内部有一个FCoE处理模块实现将CEE的流量桥接到FC交换模块中，反向过程也类似，FCoE模块的角色就是一个协议映射封装与解封模块，另外还负责对Fabric大部分逻辑的模拟。实际中FCoE功能的定价可以使用License控制，可以选择购买FCoE功能或者不购买。

这台交换机还同时可以作为传统以太网交换机使用，所以此设备是一台传统FC交换机、FCoE交换机、以太网交换机的超级融合体。图13-

12所显示的是在这种超级融合环境下的数据流拓扑图，可以看到这种设备实现了双网融合的作用。

图13-12 FCoE环境拓扑图

在网络没有融合之前，你需要投入的是：以太网卡+FC卡+FC交换机+以太网交换机+FC线缆+以太网线缆，而融合之后，你需要投入的是：CNA卡+FCoE交换机+以太网线缆，可见减少了投资，简化了布线，降低了耗电，最终也就降低了成本，同时还充分融合原有传统FC的环境。

然而，是否会降低成本也是相对的。FCoE目前实际主要应用于大型运营商、大型数据中心，准备同时投入大容量高速以太网与存储网络的，则可以在适配器、布线、交换机方面节省成本。但是FCoE离行业市场客户还差很远。16口8Gbps FC交换机也就两万元多一台，一台同样口数的10Gbps的FCoE交换机，则贵得多。但是对于运营商那种大型模块化交换机来讲，FCoE交换机由于融合了两种网络，反而比单独为每种网络都配一台大型模块化交换机要便宜多了。

如果使用纯以太网环境跑iSCSI则不可能融合原有传统FC环境了，势必又形成孤岛。FCoE达到了两者的折中，提供了一个过渡解决方案，融合了FC的思想和以太网的躯体，还融合了传统FC环境，可以说这是将单纯iSCSI击退的一个必杀技。一张FCoE卡可以同时承载普通IP以太网流量、FCoE流量以及iSCSI流量（使用软Initiator）。

目前尚未有厂商推出集硬FCoE、CEE 10GbE以太网、硬iSCSI为一身的纯硬HBA卡，Emulex公司有硬FCoE卡以及硬10GbE iSCSI卡，两种卡的设计、芯片等大致相同，有集成的潜质，但是考虑到成本、性能、

芯片等多种问题，目前这种卡可能还只是个假想物。最终鹿死谁手还有待观望。

到了这里，对之前所提出的那个疑惑是不是可以有所领悟呢？

13.13.5 解剖FCoE交换机

经过上述介绍之后，理解FCoE交换机的逻辑部件就不难了。无非就是三大主要模块：FC PHY芯片组，以太网PHY芯片组、中央交换矩阵、桥接逻辑处理芯片组，再加上一堆辅助的芯片和器件。整体上是操作系统+专用硬件芯片组的设计模式。图13-13是某FCoE交换机的主板器件布局示意图。图13-14则给出了FC与FCoE流量的处理路径示意图。

图13-13 某FCoE交换机主板器件示意图

图13-14 某FCoE交换机数据流示意图

图13-15与图13-16所示的是另外某厂商的某FCoE交换机的逻辑与实物图，板载20个CEE 10GbE接口，FCoE逻辑在Crossbar Switch中实现。每个Unified Port Controller下面挂接多个10GbE PHY。另外还有一个专门用于控制扩展板上的接口的Port Controller。

图13-15 某FCoE交换机主板器件布局图

图13-16 某FCoE交换机主板器件示意图与数据流示意图

图13-17所示为此交换机提供的6口8Gb FC接口扩展板，利用这个扩展板即可将FCoE与传统FC网络相融合了。

图13-17 某FCoE交换机的6口8Gb FC扩展接口板实物图

13.13.6 存储阵列设备端的改动

与主机相同，存储阵列也需要插上CNA卡来实现FCoE。目前Qlogic、Emulex、Brocade等主流HBA厂商都已经有了成熟的FCoE卡出售，阵列厂商的CNA卡也都是从这些HBA厂商获取。阵列端的CNA卡可以同时承载FCoE流量与iSCSI流量，也算是一种Combo卡。

13.13.7 FCoE与iSCSI

同样是oE，iSCSI如此便利，而FCoE凭什么能够兴风作浪？主要原因有两个：第一是为了融合原有的传统FC网络和FC卡件、阵列等，得有个过渡期；第二则是FC协议在上层思想上确实效率要高于TCP/IP协议，也就是高于iSCSI。但是FCoE一样还是出不去，因为FC的思想已经完全凝固了，注定了它走不出去。而iSCSI则可以随着互联网带宽的迅速增加而顺利地扩张到整个互联网范围。FCoE与iSCSI，谁O不是O？区别就是一个上层是TCP/IP，一个则是FC。iSCSI可扩展性强，但是目前来讲做硬卡的不多（Emulex和Broadcom有10Gb的iSCSI硬卡）；FCoE则高效，做硬卡的多（几乎所有做HBA的厂商都有FCoE卡产品），但是可扩展性依然不行，因为FCoE，像FC一样，很邪门，要求交换机配合，交换机里必须灌入相应的Firmware模块来处理这些Fabric协议逻辑。FCoE交换机除了需要处理Fabric逻辑，还需要处理Fabric到以太网的转换翻译逻辑，不但增加了复杂度和故障点，而且势必又会在多个厂商之间产生兼容性问题，这个恐怕会成为它称霸的一个累赘。所以我个人还是偏向于看好iSCSI的称霸。

FCoE和iSCSI只有可能是冤家不可能是朋友，不可能被融合，那么随着时间的推移，到底谁能够一统天下？或者两者各自占据自己的应用场景，还真不好说，拭目以待吧。

13.13.8 FcoE的前景

- 相对于iSCSI，FCoE能够将传统的FC网络融合，充分利用。
- 首先被大型运营商大型数据中心采用，双网融合，降低成本。
- 传统FC硬件逐渐被侵蚀殆尽。
- 以太网一统天下，将来SCSIoE、ATAoE等都有可能商业化。
- 传统FC——抛弃躯体，保留思想——FCoE。

13.13.9 Open FCoE

上文所述的所谓“软FCoE”协议，确实有这东西。Intel某高级软件工程师主导开发了它。可以访问<http://www.open-fcoe.org>这个网站来获取更多信息。这个Open FCoE协议底层可以支持普通以太网卡，但是要求以太网卡支持802.3x Pause帧。安装了软FCoE initiator的服务器可以使用其以太网卡连接到FCoE交换机上，从而被接入到传统的FC SAN中。Open FCoE也提供Target端。目前Open FCoE从Linux kernel 2.6.29开始被正式纳入。可想而知软FCoE协议栈会对服务器CPU造成多少负载增加，但是别忘了它的发起人是Intel。

观点：经过上文对FCoE的分析来看，FCoE像FC一样不开放，身体变了，思想依然没变，而且还要求身体跟着脑袋变一变（CEE/DCE）。本来承载到以太网上，指望着开放一些，现在看来，只是改朝换代而不是革命。所以，这里面的猫腻就看出来了，FCoE是当初Cisco主推的，Cisco称霸了以太网，他当然希望存储网络全都转到以太网，但是传统FC根深蒂固，所以弄出个FCoE来，以太网宁愿在FC的光辉思想下对自身的一些“瑕疵”进行修改。FCoE作为Cisco称霸存储网络的一张王牌，其目的就是为了称霸存储网络，进一步称霸数据中心，

Cisco同样拥有自己的服务器，他只是没有自己的存储，所以他选择与别人结盟。从VCE、VCN联盟就看出来了。

而Intel的Openfcoe，显然希望大家都用软Initiator，靠强大的CPU来执行而不是专用ASIC从而带动CPU的销量和发展。

那么iSCSI与FCoE到底谁能成为主流呢？现状是，iSCSI硬卡只有Emulex和Broadcom（尚未发布只是公开）的CNA卡支持，后续可能会有更多的厂商推出万兆iSCSI硬卡。如果用软iSCSI initiator的话，一个10G口子会耗掉30%左右甚至更高的CPU，就算用最新的Intel CPU也是这样，所以基本上4个10G口就会耗死一台阵列。但是FC卡就不会有这种情况，FCoE同样也不会。退一步讲，如果10G的FCoE与硬10G iSCSI比的话，卡件与协议本身来讲都差不多，成本也差不多，所以就要看外围辅助设备的成本，一台FCoE交换机目前来讲还是远贵于10G以太交换机，所以要是小规模部署，甚至不如用8G FC交换机划算了。其次是看看场景，FCoE更适合想融合之前已经部署的FC与新部署的以太网的场景，也就是大型久建的数据中心。对于新建数据中心，这个趋势还不明朗，笔者之前和Emulex的一名员工交流过，他的意思是新建数据中心他们推荐使用iSCSI硬卡，但是我估计这种说法是含有水分的，毕竟Emulex是目前仅有的一家提供硬iSCSI卡成品的公司，他们推荐iSCSI可能有一定市场目的。但是我个人看法，FCoE目前Qlogic、Brocade和Emulex都有产品了，为何10G的iSCSI硬卡只有Emulex一家产品，证明FCoE今后可能会有一波行情。FCoE和iSCSI，谁O不是O？区别就是一个是FC，一个是TCP/IP，抛开以太网，单看FC和TCP/IP，前者高效但是扩展性差，后者效率稍低但是扩展性很好，其实这已经与以太网无关了，最后到底是认同FC还是认同TCP/IP的问题。我的看法是，FCoE会弄出一波行

情，但是FC Fabric这个协议很邪门，它要求交换机也要参与Fabric的建立，而且交换机起到至关重要的作用，这增加了复杂度并且降低了兼容性，这是与IT基础架构发展背道而驰的；而iSCSI却不要求交换机有什么上层协议智能，兼容性、开放性及扩展性更好。笔者个人认为最后iSCSI很有可能会替代FCoE。但是目前来讲FCoE与iSCSI，厂商也尚未看清，谁也不敢冒然选择。

第14章 变幻莫测——虚拟化

■ 虚拟化

■ In Band

■ Out Band

“计算机科学中的任何问题，都可以通过加上一层逻辑来解决。”

——计算机科学家 David Wheeler

目前形形色色的软件层出不穷，可是它们都脱离不了一个基础，那就是计算机硬件系统。如CPU、内存和各种IO接口，以及连接它们，使它们之间相互通信的总线。

CPU内部是大量的集成逻辑电路，CPU不断受到一种电信号的“刺激”，这种刺激经过CPU内部的逻辑电路的一层层传递转换，最终输出另一种电信号。这种输入、输出动作，是有一定逻辑的。通过编写汇编代码，可以实现对CPU内部逻辑电路的刺激，并引起一系列的逻辑输出。将这些逻辑映射到人们所能理解的知识上，比如输出1代表对，输出0代表错误，或者如果输出1则继续刺激，输出0则停止刺激等，这样就构成了从基本的逻辑电路到复杂的思维逻辑的映射，由简单逻辑的层层嵌套，构成了复杂逻辑。将汇编语言，用人类容易理解的语言抽象出来，就形成了高级语言。将高级语言的意思，转换成低级语言的过程，就是编译。比如说：冬瓜，用低级语言表示冬瓜这个意思，就是“撇，横折，捺，点，点，撇……”。

虚拟化过程其实是一个由阴阳叠加而产生的一系列过程和相态。

14.1 操作系统对硬件的虚拟化

我们知道，早期的计算机系统，其实是没有操作系统的，因为操作系统本身也是靠计算机硬件执行的一种程序。操作系统就是一种可以提供给其他程序方便编写并运行的程序。由程序来运行程序，而不是由程序自己来运行，这是操作系统提供的一种虚拟化表现。

1. 早期计算机单任务模式

对于早期计算机来说，只能允许执行一个任务，整个计算机只能被这个程序独占。比如开机，从软盘或者其他介质上执行程序，直到执行完毕或者人为中断。执行完后拿出介质，再插入另一张介质，重新载入执行另一个新的程序。在执行程序的过程中，一旦意外终止，就要重新运行。

如果有10个人要用一台计算机来执行程序，第一个人拿着他的软盘，上面有一个数学题计算程序，他插入软盘，然后重启机器，机器从软盘特定的扇区载入程序代码执行，结果显示在显示器上，比如这个程序2个小时运行完毕，第一个人从显示器上抄下结果，走了。后面有9个人在排队等待用计算机。然后第二个人同样拿着他的软盘，插入软驱，重启.....每次更换程序，都需要重新启动机器，简直就是梦魇。再者，如果某个程序运行期间，会有空闲状态，则其他程序也仍然需要等待，CPU只能在那里空振荡。

2. 操作系统的多任务模式

操作系统的出现解决了这两个问题。操作系统本身也是一段程序，

计算机加电之后，首先运行操作系统，随时可以载入其他程序执行，也就是说，它可以随时从软盘上读取其他程序的代码，并切换到这段代码上让CPU执行，执行完毕后则立即切换回操作系统本身。但是每次也总是要等待这个程序执行完毕，才能接着载入下一个程序执行。当被载入的程序执行的时候，不能做任何其他的事情，包括操作系统本身的程序模块，任何产生中断的事件，都会中断正在运行的程序。

程序执行完毕之后，会将CPU使用权归还操作系统，从而继续操作系统本身的运行。这种操作系统称为单任务操作系统，典型代表就是DOS。

一旦在DOS中载入一个程序执行，如果没有任何中断事件发生，则这个程序就独占CPU，执行完毕之后，回到DOS操作系统，接着可以继续执行另外一个程序。经过这样的解决，执行多个程序，期间就再也不用重新启动机器了。

在这个基础上，操作系统又将多个程序一个接一个地排列起来，成批地执行，中途省掉了人为载入程序的过程，这叫做批处理。批处理操作系统，相对于单任务操作系统来说，可以顺序地、无须人工干预地批量执行程序，比简单的单任务操作系统又进了一步，但是其本质还是单任务性，即一段时间之内，仍然只会观察到一个应用程序在运行，仍然只是一个程序独占资源。

再后来，操作系统针对系统时钟中断，开发了专门的中断服务程序，也就是多任务操作系统中的调度程序。时钟中断到来的时候，CPU根据中断向量表的内容，指向调度程序所在的内存地址入口，执行调度程序的代码，调度程序所做的就是将CPU的执行跳转到各个应用程序所在的内存地址入口。每次中断，调度程序以一定的优先级，指向不同程

程序的入口。这样就能做到极细粒度的应用程序入口切换，如果遇到某个程序还没有执行完毕就被切出了，则操作系统会自动将这个程序的运行状态保存起来，待下次轮到的时候，提取出来继续执行。比如每10ms中断一次，那么也就是说每个应用程序，可以运行10ms的时间，然后CPU运行下一个程序，这样依次轮回。微观上，每个程序运行的时候，还是独占CPU，但是这个独占的时间非常小，通常为10ms，那么一秒就可以在宏观上“同时”运行100个程序。这就是多任务操作系统。多任务操作系统的关键，就是具有多任务调度程序。

通过这样的虚拟化，运行在操作系统之上的所有程序都会认为自己是独占一台计算机的硬件运行。

3. 虚拟化的好处

上面介绍了计算机硬件以及操作系统，其实计算机系统从诞生的那一天开始，就在不断地进行着虚拟化过程，时至今日，计算机虚拟化进程依然在飞快发展着。

硬件逻辑被虚拟化成汇编语句，汇编语句再次被封装，虚拟化成高级语言的语句。高级语言的语句，再次被封装，形成一个特定目的的程序，或者称为函数，然后这些函数，再通过互相调用，生成更复杂的函数，再将这些函数组合起来，就形成了最终的应用程序。程序再被操作系统虚拟成一个可执行文件。其实这个文件代表了什么呢？到了底层，其实就是一次一次的对CPU的电路信号刺激。也就是说，硬件电路逻辑，一层层地被虚拟化，最终虚拟成一个程序。程序就是对底层电路上下文逻辑的另一种表达形式。

虚拟化的好处显而易见，虚拟化将下层的复杂逻辑转变为上层的简

单逻辑，方便人类读懂，也就是说“科技，以人为本”。任何技术，都是为了将上层逻辑变得更加简单，而不是越变越复杂，当然使上层越简单，下层就要做更多的工作，就越复杂。

整个计算机技术，从开始到现在，就是一个不断地抽象、封装、虚拟、映射的过程，一直到现在还在不断抽象封装着，比如Java等比C抽象封装度更高的高级语言，当然使用起来也比C方便和简单多了，但是随之而来的，其下层就要复杂一些，所以Java代码一般运行速度慢，耗费资源也大，但是对于现在飞速发展的硬件能力，是不成问题的。

同样，CPU也不仅仅只是一味地增加晶体管数量这么简单。CPU制造者也在想尽办法将一些功能封装到CPU的逻辑电路中，从而出现了更多的指令集，这些指令集就像程序函数一样，不必理解它内部到底怎么实现的，只需要发给CPU，CPU就会启动逻辑电路计算。到目前为止，Intel的CPU已经发展到了酷睿多核。主频1.6GHz的酷睿四核CPU，性能毫无悬念地比主频3GHz甚至超频到4GHz的奔腾4代CPU还高。所以CPU的设计除了提高主频之外，更重要的是内部逻辑的优化，集成度的提高，更多的抽象和封装。

提示： 虚拟化的思想在计算机的各个方面都是存在的，比如经典的OSI模型，就是一个不折不扣的抽象虚拟模型。尤其是TCP/IP协议给上层抽象出来的Socket接口，即“插座接口”，也可以理解为，只要将插头接上这个插座，就会和网络接通，而不必管它是怎么实现的，就像将交流电插头插入市电插座一样，插上就获得了电压，而不用管市电电网的拓扑，更不用担心国家总电网的拓扑了。

14.2 计算机存储子系统的虚拟化

上面介绍了很多关于汇编和操作系统的虚拟抽象，下面将介绍计算机系统中的存储子系统中的虚拟化。

存储子系统的元素包括磁盘、磁盘控制器、存储网络、磁盘阵列、卷管理层、目录虚拟层及文件系统虚拟层。下面从下到上，一一描述这几个元素，看看存储子系统是怎么抽象虚拟的。

1. 磁盘控制器

磁盘控制器的工作就是根据驱动程序发来的磁盘读写信息，向磁盘发送SCSI指令和数据。这个部件看似没有什么可抽象虚拟的东西，其实磁盘控制器完全可以对其驱动程序隐藏其下挂的物理磁盘，而虚拟出一个或者多个虚拟磁盘。由控制器来完成虚拟磁盘和物理磁盘的映射和抽象虚拟。RAID就是一个典型代表，控制器将物理磁盘组成RAID Group，然后在RG的基础上，虚拟出多个LUN，通告给主机驱动。

2. 存储网络

早期的存储子系统，没有网络化；而目前的存储系统，网络化已经非常彻底。从磁盘到磁盘阵列控制器，从磁盘阵列控制器到主机总线适配器，都已经嵌入了网络化元素。比如使用FC协议，或者TCP/IP协议、SAS协议、Infiniband协议等。那么在这一层上，有什么可以抽象的吗？网络化只是为部件之间提供了一种可扩展的传输通路而已，貌似在这个层面上不能做出什么大文章来。

实则不然，这一层也是有所深究的。在交换式SAN中，不管是基于TCP/IP协议的还是基于FC协议的SAN，网络中的任何节点，都是通过交换设备来互相通信，这是节点间通信的必经之路。如果在交换设备上做点手脚，就完全可以达到虚拟化的效果。

要抽象一种逻辑，那么一定要理解这种逻辑，所以我们可以FC交换机或者以太网交换机上，嵌入SCSI协议感知模块。比如某个N节点向另一个N节点Report LUN的时候，交换机收到这个Frame，则可以感知这个N节点的LUN信息。如果此时网络中还有另一个节点的LUN信息，则可以在交换机这一层到达这两个节点的LUN的镜像。也就是说，SCSI发起设备向目标设备传输的数据，经由交换机的时候，交换机内嵌的虚拟化模块，会主动复制对应的帧到另一个节点的LUN上，让这两个LUN形成镜像，当其中一个节点故障的时候，交换机因为知道此时还有一个备份镜像LUN存在，所以并不会向发起者通告失败，而是默默地将发起者的数据重定向到这个镜像的LUN，发起设备并不会感知，这样，就达到了基于网络层的虚拟化抽象。

当然，网络层的虚拟化并不只是镜像，是将某些N节点的LUN合并成一个池，然后动态地从这个池中再划分出虚拟LUN，向发起者报告等。基于这些思想，已经开发出了智能FC交换机。

3. 磁盘阵列

磁盘阵列可以说本身就是一个小计算机系统（JBOD除外），这个系统五脏俱全，是对存储子系统的抽象虚拟化最佳的表现。磁盘阵列，简要地说，就是将大量磁盘进行组织管理，抽象虚拟，最终形成虚拟的逻辑磁盘，最后通过和主机适配器通信，将这些逻辑磁盘呈现给主机。这个功能和前面提到的磁盘控制器的功能类似，但是磁盘阵列能比狭义

的磁盘控制器提供更多的特色功能，况且简单地插在主机IO总线上的那种RAID磁盘控制器，其接入磁盘数量有限，功能也有限。

大型磁盘阵列，有自己的控制器，有的利用嵌入式技术，将特别定制的操作系统及其核心管理软件嵌入芯片中，来管理整个控制器并实现其功能；有的则干脆利用现成的主机来充当盘阵控制器的角色，比如IBM的DS8000系列盘阵，内部就是用的两台IBM P系列小型机作为其组织管理磁盘的控制器，其上运行AIX操作系统和相应的存储管理软件。

不管是嵌入式，还是主机式的，盘阵控制器所担任的角色都是类似的。这个中心控制器，不直接参与连接每块磁盘，而是利用后端适配器来管理下挂的磁盘，由后端适配器向其上级汇报。

这些适配器，就是由中心控制器驱动的二级磁盘控制器，这些磁盘控制器作为中心CPU的IO适配器，直接控制和管理物理磁盘，然后由中心控制器统一实现RAID、卷等高级功能（有些盘阵则可以将简单的RAID功能直接下放给二级控制器来做）。后端适配器与中心控制器CPU之间通过某种总线技术连接，如PCIX、PCIE总线等。中心控制器对这些磁盘进行虚拟抽象之后，通过前端的接口，向最终使用它的主机进行通告。中心控制器不但可以实现最基本的RAID功能，而且可以实现很多高级功能，如LUN镜像、快照、远程复制、CDP数据保护、LUN再分配等。在磁盘阵列上实现虚拟化，是目前最广泛的一种存储系统虚拟化形式。

有些产品甚至学成了借花献佛的本领。比如NetApp公司V系列NAS网关、HDS公司的某些存储设备以及IBM公司的SVC。这些设备面对后端存储时，它就是主机，而面对前端主机的时候，它们就是存储，如图14-1所示。

图14-1 借花献佛

提示：乍一感觉这台虚拟化设备也真够无赖的，明明自己没有磁盘却能踩着别人的脚向外提供LUN，明明就是自己向别人租赁来的然后又装修了一把，转租出去。但是我们非常需要这种设备，因为它帮了大忙。

假如，图14-1中磁盘阵列1的容量为1TB，磁盘阵列2的容量为1TB，而某台主机需要一个容量为2TB大小的LUN，这怎么办呢？我们可以在主机上安装卷管理软件，让VM把这两个1TB的LUN合并成一个2TB的卷即可。但是这么做需要耗费主机资源，且虚拟好的新LUN只能给这台主机使用。而一些旧的低端设备，由于其容量和性能等已经不能满足要求，如果可以将这些设备挂到这台虚拟化设备上，作为一个二线存储资源，这样就将所有的存储资源整合到了一起，统一管理和分配。

要想获得足够的性能和灵活性，就需要图14-1所示的虚拟化设备了。

这个设备在盘阵端（后端）的FC HBA卡处于FC Initiator模式，即在后端，这台设备以主机模式出现。而在前端（主机端），这台设备的FC HBA卡为FC Target模式，即它以盘阵的角色出现。这样，这台设备就可以从其后端掌管LUN，然后将这些LUN合并并再次灵活分割，呈交给其前端的多台主机使用。除了简单的合并再分割LUN之外，这台设备还可以做许多其他数据管理操作，比如将两个LUN镜像、快照、CDP等。

4. 卷管理层

卷管理层是指运行在应用主机上的功能模块。它负责底层物理磁盘或者LUN的收集和再分配。经过盘阵控制器虚拟化之后生成的LUN提交给主机使用，主机可以对这些LUN进行再次抽象和虚拟，也就是重复虚拟化，比如对其中两个LUN进行镜像处理，或者对其中的多个LUN，做成一个软RAID系统。再或者将所有LUN合并，形成一个大的资源池，然后像掰面团一样掰成多个卷，这个过程和磁盘控制器、盘阵控制器所做的虚拟化动作类似，但是这个动作是在主机上实现的。典型的卷管理软件有LVM，或者第三方的软件，比如Veritas公司的VxVM。

5. 文件系统

数据只是存储到磁盘上就完了吗？显然不是。打个比方，有位记者早晨出去采访，手中拿了一摞纸，他每看到一件事就记录下来。对于“怎么将字写在纸上”这个问题，他是这么解决的，他用笔在格子上写字，写满一行再写下一行，还不够就换一张纸。对于“怎么让自己在纸上写字”这个问题，是他自己通过大脑（控制器），通过神经网络（SCSI线缆），操纵自己的手指（磁头臂），拿着笔（磁头），看见有格子，就向里写。这两个问题都解决了。可是这一天下来，他回去想看看一天都发生了什么，他拿出记录纸，却发现，信息都是零散的，根本无法阅读，有时候读到一半，就断了，显然当时是因为格子不够用了，写到其他地方了，造成了信息记录的不连续，有的地方还有删除线，证明这一块作废了，那么有效的记录到底在哪里呢？记者方寸大乱，数据虽然都完好地记录在纸上，但是它们都是不连续的、凌乱的，当时是都记下来了，但是事后想要读取时却没辙了。

磁盘记录也一样，只解决磁盘怎么记录数据和怎么让磁盘记录数据，是远远不够的，还应该考虑“怎么组织磁盘上的数据”。

还是用这个记者的例子来说明。我们都能想到，将凌乱的记录组织成完整的一个记录，只需要在相应的地方做一下标记，比如“此文章下一段位于某某页，第几行”，就像路标一样，一次一次地指引你最终找出这个完整的数据，这个思想称为“链表”。

如果将这个链表单独地做成一个记录，存放到固定位置，每次只要参考这个表，就能找出一条数据在磁盘上的完整分布情况。利用这种思想做出来的文件系统，比如FAT文件系统，它把每个完整的数据称为文件。文件可以在磁盘上不连续地存放，由单独的数据结构来描述这个文件在磁盘上的分布，这个数据结构就是文件分配表。File Allocate Table，也就是FAT的由来。或者用另一种思想来组织不连续的数据，比如NTFS，它是直接给出了一个文件在磁盘上的具体扇区，开始—结束，开始—结束，用这样的结构来描述文件的分布情况。

文件系统将磁盘抽象成了文件柜，同一份文件可能存放在一个柜子的不同抽屉中，利用一份特别的文件来记录“文件一对应抽屉”的分布情况，这些用来描述其他文件分布情况及其属性的文件，称为元文件（Metadata）。元文件一般情况下要存放在磁盘的固定位置，而不能将其分散，因为最终要有一个绝对参考系统。但是有些文件系统，甚至将元文件也可以像普通文件一样，在磁盘上不连续地分布。前面还说过一定要有一个绝对参考系统，也就是固定的入口，所以这些特殊的文件系统，其实最上层还是有一个绝对参考点的，这个参考点将生成元文件/在磁盘上的分布情况记录，从而定位元文件，再根据元文件，定位数据文件，这样一层一层地嵌套，最终形成文件系统。

最终一句话，文件系统是对磁盘块的虚拟、抽象、组织和管理。用户只要访问一个个的“文件”，就等于访问了磁盘扇区。而访问文件，这个动作是非常容易理解的，也是很简单的，用户不必了解这个文件最终

在磁盘上是存放到哪里，怎么存放的，怎么访问磁盘来存放这个文件，这些统统都是由文件系统和磁盘控制器驱动程序来做。

6. 目录虚拟层

不管是Windows系统、UNIX系统，还是Linux系统，其内部都有一个虚拟的目录结构。在Linux中叫做VFS，即Virtual File System。

虚拟文件系统，顾名思义也就是说这个文件系统目录并不是真实的，而是虚拟的。任何实际文件系统，都可以挂载到这个目录下，真实FS中的真实目录，被挂载到这个虚拟目录下之后，就成为了这个虚拟目录的子目录。这样做的好处是增强灵活性。其次，操作系统目前处理外部设备，一般都将其虚拟成一个虚拟文件的方式，比如一个卷，在Linux中就是/dev/hda这种文件。对这个文件进行读写，就等于直接对设备进行了读写。

存储子系统的虚拟化，可以在“磁盘—盘阵控制器—存储网络—主机总线适配器—卷管理层—文件系统层—虚拟目录层和最终应用层”各个环节虚拟抽象地工作，使得最终应用软件，只要通过文件系统访问文件，就可以做到访问最底层的磁盘一样的效果。有时候还可以重复虚拟化。

14.3 带内虚拟化和带外虚拟化

所谓带内即In Band，是指控制信令和数据走的是同一条路线。所谓控制信令，就是说用来控制实际数据流向和行为的数据。典型的控制信令，比如IP网络中的各种IP路由协议所产生的数据包，它们利用实际数据线路进行传输，从而达到各个设备之间的路由统一，这就是带内的概念。

带外即Out Band，是指控制信令和实际数据走的不是同一条路，控制信令走单独的通路，受到“优待”。

带内和带外，只是一种叫法而已，在电话信令中，带内和带外是用“共路”和“随路”这两个词来描述的。共路信令指的是控制信令和实际数据走相同的线路；随路信令则指二者走不同的线路，信令单独走一条线路。随路又可以称作“旁路”，因为它是单独一条路。

明白了上面这些概念，用户就可以理解所谓“带内虚拟化”和“带外虚拟化”的概念了。

带内虚拟化，就是说进行虚拟化动作的设备，是直接横插在发起者和目标路径之间的，斩断了二者之间的通路，执行中介操作，发起者看不到目标，而只能看到经过虚拟化的虚拟目标。所以在带内虚拟化方式下，数据流一定会经过路径上的所有设备，即所有设备是串联在同一条路径上的，虚拟化设备插入这条路径中，作为一个“泵”，经过它的时候就被虚拟化了。

带外虚拟化，则是在这个路径旁另起一条路径，也就是所谓旁路。用这条路径来走控制信号，而实际数据还是由发起者直接走向目标。但

是这些数据流是受控制信令所控制的，也就是发起者必须先“咨询”旁路上的虚拟化设备，经过“提示”或者“授权”之后，才可以根据虚拟化设备发来的“指示”直接向目标请求数据。带外虚拟化方式中，数据通路和信令通路是并联的。

带内虚拟化的例子非常多，目前的虚拟化引擎几乎都是带内虚拟化。IBM的SVC（San Volume Controller）、Netapp的V-series、HDS公司的USP系列等，它们都是带内虚拟化引擎。

1. 带外虚拟化系统SanFS

带外虚拟化的一个典型的例子，是IBM公司的SanFS系统。图14-2显示了SanFS的基本架构。

图14-2 IBM公司的SanFS架构示意图

SanFS其实根本没有什么高深的地方，说白了，SanFS就是一个网络上的文件系统，也就是说，常规的文件系统都是运行在主机服务器上的，而SanFS将它搬到了网络上，用一台专门的设备来处理文件系统逻辑。

然而，这个“网络上的文件系统”却绝对不是“网络文件系统”。网络文件系统是典型的带内虚拟化方式，因为网络文件系统对上层屏蔽了底层卷，只给上层提供一个目录访问接口，上层看不到网络文件系统底层的卷，只能看到目录。而SanFS架构中，上层既能看到文件系统目录，又能看到底层卷（LUN），如图14-3所示。

图14-3 SanFS架构示意图

图14-3中，左边的服务器是一台普通的服务器，右边是一台使用SanFS文件系统的服务器。右边的服务器上的文件系统已经被搬到了外面，即运行在一台SanFS控制器上，这个控制器与服务器都接入一台以太网交换机。当虚拟目录层需要与文件系统层通信的时候，通信路径不再是内存，而是以太网了。由于文件系统已经被搬出主机，所以任何与文件系统的通信都要被重定向到外部，并且需要用特定的格式，将请求通过以太网发送到SanFS控制器，以及从控制器接受相应的回应，所以在使用SanFS的服务器主机上，必须安装SanFS管理软件（或者叫做SanFS代理）。

下面用几个实例来说明SanFS是如何作用的。

实例1

服务器运行Windows 2003操作系统，使用SanFS作为文件系统的卷的盘符为“S”盘。在Windows中双击盘符S，此时VFS虚拟目录层便会发起与SanFS控制器的通信，因为需要获取盘符S根目录下的文件和目录列表。所以VFS调用SanFS代理程序，通过以太网络向SanFS控制器发送请求，请求S根目录的文件列表，SanFS控制器收到请求之后，将列表通过以太网发送给SanFS代理，代理再传递给VFS，随即就可以在窗口中看到文件和目录列表了。

实例2

某时刻，某应用程序要向S盘根目录下写入一个大小为1MB的文件。VFS收到这个请求之后，立即向SanFS控制器发送请求，SanFS控制器收到请求之后，计算出应该使用卷上的哪些空闲块，将这些空闲块的LBA号码列表以及一些其他必要信息通过以太网传送到服务器。服务器

上的SanFS收到这些信息后，便调用操作系统相关模块，将应用的数据从服务器的内存中直接向下写入对应卷的LBA地址上。

SanFS系统是一个典型的带外虚拟化系统，服务器主机虽然可以看到底层卷，但是管理这个卷的文件系统，却没有运行在主机上，而是运行在主机之外。主机与这个文件系统之间通过前端以太网通信，收到文件系统的指示之后，主机才按照指示将数据直接写入卷。

SanFS究竟有何意义呢？SanFS是不是有点多此一举呢？放着主机内存这么好的风水宝地不用，却自己跑出去单独运行，和别人通信还得忍受以太网的低速度（相对内存来说），这是何苦呢？煞费苦心的SanFS当然有自己的算盘，这么做的原因如下。

将文件系统逻辑从主机中剥离出来，降低主机的负担。

既然将文件系统从主机剥离出来，为何不干脆做成NAS呢？NAS同样也是将文件系统搬移出主机。答案是因为向NAS传输数据，走的是以太网，速度相对FC SAN要慢。所以SanFS的设计是只有元数据的数据流走以太网，实际数据依然由主机自行通过SAN网络写入盘阵等存储设备。这样就加快了数据的传输速度，比NAS有优势。

其实SanFS一个最大的特点，就是支持多台主机共享同一个卷，即同一时刻可以由多台主机共同读写（注意是读写）同一个SanFS卷。这也是SanFS最大的卖点。共享同一个LUN的所有主机，都与SanFS控制器通信以获得访问权限，所以SanFS干脆就自己单独占用一台专用设备，放在网络上，也就是SanFS控制器，这样可以让所有主机方便地与它连接。

2. SanFS与NAS的异同

SanFS与网络文件系统究竟有何不同呢，如图14-4所示。

图14-4 SanFS与NAS的异同

显然，NAS（网络文件系统）与SanFS是截然不同的。主机向NAS写入数据，其实要经历两次写的过程，第一次写是主机将数据通过以太网发送给NAS的时候，第二次写是NAS将收到的数据写入自己的硬盘（本地磁盘或者SAN上的LUN卷）。而SanFS只写入一次数据，而且是通过FC SAN，而不是相对慢速的以太网。

3. 轮回和嵌套虚拟化

以太网在传统上是用来承载IP的，但是有一些技术是将以太网承载到IP之上的，比如VPLS。VPLS属于一种极端变态的协议杂交方式。它嵌套了多次，也轮回了多次。

VPLS可以说是对以太网VLAN技术的一种上层扩展。传统VLAN使用VLAN标签来区分不同的域，VPLS则可以直接通过用不同的IP来封装以太网头来区分各个以太网域。

这个技术也从一个侧面表明了TCP/IP在当今网络通信领域所不可动摇的绝对地位。

14.4 硬网络与软网络

1. 硬件网络设备

所谓硬件网络设备，其功能终究还是靠软件来实现的。很多网络硬件设备，尤其是路由设备，本质上就是一台PC或者PC Server。其上运行着专门处理网络数据包的程序。就这样，若干底层网络设备互相连接，组成了整个基础网络，也就是硬件网络环境。

在硬件网络环境的基础上，若干PC接入硬件网络，实现相互通信。也就是说，用一部分PC充当网络硬件设备，其他PC利用这些充当网络设备的PC实现通信。这就是一种嵌套的表现，也就是“网中有网”。

2. 软件网络程序

Message Queue（MQ）和Message Broker（MB）在硬件网络设备的基础上，模拟出一个纯软件的网络转发引擎。这就是一种轮回的表现。

MQ是一种消息转发软件引擎。这个引擎运行在主机操作系统之上。其功能就是充当一个消息转发器。客户端通过TCP/IP与这个转发器相连，将消息传送到这个转发器上，然后转发器根据策略，将消息转发到其他客户端上。这种消息转发器，也就类似于网络交换机。只不过MQ的链路层由TCP/IP来充当。

MB是一种应用逻辑转发引擎。这个引擎虽然也是用来转发消息的，但是它不仅仅是底层转发，还能做到应用层次的转发。这类似于邮件服务器，只不过它可以转发各种格式和方式的数据包。

14.5 用多台独立的计算机模拟成一台虚拟计算机

1. HPC环境

点组成线，线组成面，面组成体，体与体之间组成网，然后就是进化。同样，HPC环境也是这种模式。在一个典型的HPC环境中，包含众多的计算机，这些计算机各有分工。总体来说，HPC环境中的计算机可以分为两大类：一种是专门用来计算数据的，为CPU密集运算；另一种是专门用于存储计算过程中，所需要提取或者存放数据的，为IO密集运算。前者称为计算节点，后者称为存储节点。而为了最大利用硬件资源，有些HPC环境中会存储节点，也兼用来做计算节点。

可以将一个HPC环境中的所有计算节点看作一台大的虚拟计算机的CPU和内存，而将所有存储节点看作虚拟计算机的硬盘。虚拟计算机的CPU和内存（计算节点），通过某种连接链路向虚拟计算机的硬盘（存储节点）读写数据，从而计算出结果。对于一台单独的物理计算机来说，CPU内存与存储设备之间的连接为高速IO总线，比如PCIE。但是对于由多台独立节点组成的HPC系统来说，虚拟CPU与虚拟存储设备之间的连线就不可能是内部IO总线了，而是一种外部的高速网络传输方式。有些HPC利用Infiniband网络作为计算节点与存储节点之间的连接方式，有些则干脆使用以太网。前者一般用于IO密集型的运算；后者一般用于CPU密集型运算，也就是说，运算过程中需要读写的数据不多。

2. 典型的Web+APP+DB架构

这种架构是一种典型的IT架构。客户端通过Web服务器获取一个图形化显示网页，应用逻辑由APP（Application）服务器处理，并将结果

通过Web服务器显示到客户端的网页上，APP服务器需要的数据则通过访问数据库服务器来获得。

也可以将Web服务器看作一台显示终端，将APP服务器看作CPU和内存，将DB服务器看作硬盘。这样，一个由Web+APP+DB服务器所组成的虚拟计算机便诞生了。

14.6 用一台独立的计算机模拟出多台虚拟计算机

1. VMware虚拟机软件

VMware通过模拟一套硬件系统，将程序对这个硬件系统CPU发送的指令经过一定的处理之后，并加以虚拟传到物理CPU上执行。利用这种方式，可以在一台物理计算机上虚拟出多个虚拟机。

目前Windows Server 2008操作系统已经自带了HyperV虚拟化引擎。类似VMware的ESX。目前很多操作系统都集成了Native的虚拟化引擎。

2. 世界本身就是一个轮回嵌套的虚拟化系统

不但在计算机领域中有虚拟化，在其他学科中同样有虚拟化。在化学领域中，科学家把观察到的现象和计算出来的公式，虚拟化成原子和分子。

总之，一切都是虚拟化的结果，我们观察到的世界其实就是我们利用基本数学公式虚拟出来的。人们首先在大脑中演绎出数学，然后虚拟化出了物理学，然后再用数学和物理学虚拟化出化学等其他各种学科。这就像用汇编语言来抽象数字电路逻辑，再用高级语言来抽象汇编语言，然后将现实中的逻辑用计算机高级语言表达出来，让计算机来模拟出现实逻辑。

14.7 用磁盘阵列来虚拟磁带库

VTL，即Virtual Tape Library，虚拟磁带库。传统的物理磁带库为全机械操作，比如机械手、驱动器、磁带等。其速度相对磁盘来说要慢很多，如果需要备份的数据量非常大，而备份窗口又很小，那么只能通过提高磁带库的速度来解决。但是要提高磁带库的速度，只能同时用多个驱动器同时操作，需要成本高，不方便。虚拟磁带库的出现为的就是解决上述这些问题。VTL使用磁盘来存储数据而不是磁带，并虚拟出机械手、磁带驱动器、磁带这三样在物理上都不存在的东西。在备份软件等使用磁带库的应用程序，不会发现物理设备到底是盘阵还是真实的磁带库。而虚拟化之后，前端的程序接口不变，后端的速度和灵活性却大大增加了。

图14-5是一台物理磁带库的正视图。图14-6显示了仓门打开后其内部构件示意图，可以看到一根竖直的柱子，这个柱子就是机械手的滑轨。机械手可以沿着柱子上下滑动并且可以左右转动，以抓取右侧磁带槽中的磁带。图14-7所示是物理磁带库的两个驱动器和电源后视图。在图14-8中可以看到另一种设计的机械手和驱动器。

图14-5 物理磁带库的正视图

图14-6 物理磁带库的内视图

图14-7 物理磁带库后视图（驱动器和电源）

图14-8 另一种设计方式的磁带库机械手

NetApp VTL700配置使用实例

比如有NetApp VTL700系列虚拟磁带库一台，机头连接了两个扩展

柜。共28块500GB的SATA磁盘。

(1) 第一步：创建**RAID**组，为虚拟磁带创建底层存储空间。

磁盘阵列必须做**RAID**，这是任何情况下都要保证的。**RAID**不仅可以提高速度，更重要的是为了保护数据，因为任何一块硬盘损坏，如果没有**RAID**，都会造成数据丢失。**VTL**使用的磁盘阵列也不例外。

VTL700可以利用**Web**界面来管理，管理主界面如图14-9所示。

图14-9 VTL700管理主界面

查看当前系统的虚拟磁带容量等信息，如图14-10所示。由于还没有配置完成，所以图14-10中没有给出任何虚拟之后的容量等信息（**RAID**组和虚拟磁带总容量都为0.00Gb）。

图14-10 系统配置容量信息

从页面左侧栏中选择**RAID Groups**标签，右侧会显示出当前系统中所配置的**RAID**组，如图14-11所示。由于当前还没有配置任何**RAID**组，所以两台扩展柜中的所有磁盘均显示为灰色（空闲状态），蓝色的磁盘为**Spare**盘。

图14-11 系统当前**RAID**组信息

选中图14-11上方的**Create Raid Group**复选框，出现如图14-12所示的页面。

图14-12 创建**RAID**组页面

单击图14-12中的**Automatically**

图14-13 创建**RAID**组规则

Create RAID Group按钮，系统自动创建RAID组。随后出现如图14-13所示的窗口，提示用户创建RAID组的规则。

单击Apply按钮，出现如图14-14所示的页面，可以看到系统已经创建好了一系列的RAID组，且每个RAID组用不同颜色表示。

图14-14 系统自动创建的6个RAID组

单击Drop RAID Group标签，进入删除RAID组页面，可以删除已经创建好的RAID组，如图14-15所示。

图14-15 删除RAID组页面

单击Drop按钮，删除对应的RAID组。删除之后也可以再次手动或者自动创建RAID组。图14-16所示为手动创建的7个RAID组。

图14-16 手动创建的7个RAID组

回到Monitor页面，查看系统当前的配置容量信息，RAID Group栏目中已经显示出了系统当前的可用磁盘容量，如图14-17所示。

图14-17 RAID组配置完后的系统容量信息

（2）第二步：创建虚拟磁带库和虚拟磁带。

创建完RAID组之后，就为系统提供了基本的存储空间。在这个存储空间之上，还需要创建一台或者多台虚拟带库，然后创建虚拟磁带。

创建虚拟磁带库。如图14-18所示创建一台名为lib1的虚拟带库，带库的厂家信息为NetApp。虚拟磁带驱动器类型为IBM LTO1，这个类型必须对应一款物理磁带驱动器，下拉列表框中有多种支持的物理驱动器

可选。Slot Count即磁带槽位，这里定为8。Drive Count即驱动器数量，这里定为2。Assigned to Port表示将这个虚拟磁带库在哪个前端端口“提交”出去，即主机端可以从VTL上的哪个前端端口识别到这台虚拟磁带库。Fully Loaded with Virtual Tapes表示是否自动创建对应的虚拟磁带并插满磁带槽（磁带槽和磁带都是虚拟的）。Start Tape Label表示起始标签名称。物理磁带库中，每盘磁带都有各自的标签，机械手通过这些标签来识别每一盘磁带，虽然VTL不需要用标签来识别每个虚拟磁带，但是备份软件等程序需要知道这些标签，所以VTL还是要给每个虚拟磁带分配标签。这里用“L1”来起始，这样，第二盘虚拟磁带系统就会自动为其分配“L2”的标签；依此类推。

图14-18 创建虚拟磁带库

单击Apply按钮之后，一台虚拟磁带库就创建完并可以使用了。此外，还可以创建更多的虚拟磁带，如图14-19所示。单击左侧栏中的Virtual Tapes标签，右侧页中可以选择新磁带归属于哪个虚拟带库。这里选择刚刚创建的lib1，然后起始标签设为M1，数量为8盘，单击Apply按钮。这样就向lib1这台虚拟带库中增加了8盘磁带，加上原有的一共16盘。

图14-19 增加虚拟磁带

单击左侧栏中的Virtual Libraries标签，右侧会出现一张虚拟磁带库拓扑图。如图14-20所示，右侧机柜中有8个磁带槽和2个驱动器，与创建带库时的选项一一对应。左边的推车上还有8盘磁带，这是刚才追加的8盘磁带，而虚拟带库中的磁带槽放不开，所以就放到了虚拟推车上。推车上方是一个存储箱，可以用鼠标把任意一盘磁带拖动到推车、磁带槽或者驱动器中。如果拖动到存储箱中，则这盘虚拟磁带便被删除

了（如图14-21所示）。

图14-20 当前虚拟磁带库的示意图页面

图14-21 删除虚拟磁带

返回到Monitor页面中可以看到当前系统中的虚拟带库信息，如图14-22所示。

图14-22 当前系统的虚拟带库信息

创建第二台虚拟带库。再次创建一台名为lib2的虚拟带库。这里选择驱动器类型为HP LTO1，16槽位，4驱动器，如图14-23所示。图14-24显示了lib2的拓扑图，可以和lib1的图对比一下。

图14-23 创建lib2虚拟带库

图14-24 lib2虚拟带库拓扑图

单击左侧栏中的Virtual Tapes标签，在右侧页面中可以查看虚拟磁带的信息，修改虚拟磁带的属性，在多台虚拟带库中移动磁带，删除虚拟磁带，如图14-25～图14-28所示。

图14-25 显示虚拟磁带信息

图14-26 修改虚拟磁带的读写属性

图14-27 在多台虚拟带库之间移动磁带

图14-28 删除虚拟磁带

返回Monitor页面可查看系统中的信息，此时出现另一台虚拟带库，如图14-29所示。

图14-29 系统中的两台虚拟带库

手动将磁带放入驱动器。如图14-30所示，可以在拓扑图中拖动任

何一盘磁带进入驱动器。这个动作就好比在物理带库中，由机械手将磁带从磁带槽中抓出，并推入磁带驱动器。当然，备份软件可以发送指令让带库自动做这个动作，当然也可以手动做这个动作。磁带放入驱动器之后，Monitor页面中会显示出当前带库的驱动器中所包含的磁带，如图14-31所示。

图14-30 手动将磁带放入驱动器

图14-31 Monitor页面中显示的驱动器状态

（3）第三步：在客户端使用虚拟磁带库。

客户端对VTL的使用与使用一台纯物理磁带库没有任何区别。我们用一台NetApp的FAS3050磁盘阵列来识别这台VTL，看看效果。在FAS3050命令行中输入`sysconfig -a`来查看当前系统中的所有设备，可以看到在FC通道0c上已经识别到了两个IBM的驱动器和4个HP的驱动器，这和VTL700上的配置完全一样，如图14-32所示。输入`storage show tape`命令可以查看更详细的磁带驱动器信息，如图14-33所示。输入`storage show mc`命令可以查看识别到的机械手信息（机械手是VTL虚拟出来的），如图14-34所示。输入`sysconfig -t`命令可以查看系统识别到的所有磁带，如图14-35所示。

图14-32 客户端所识别到的磁带设备

14-34 查看机械手信息

图14-33 详细的驱动器信息图

图14-35 系统所识别到的所有磁带

至此，这台VTL虚拟出了两台带库，当然还可以虚拟更多的带库，将它们分配到另外的FC端口。更加灵活的是，VTL还可以自身连接物理磁带库，然后将这些物理资源透传到主机端，这样即使原来存在的物理

带库也没有浪费，一起整合了进来。

各个厂家的VTL产品的设计都是大同小异，几乎都是用各自已经成形的盘阵产品，将其上运行的程序换一下，就变成了VTL。图14-36是EMC公司的VTL产品的配置界面。可以看到各个厂家的设计都大同小异，本质都是一样的。

图14-36 EMC公司的VTL产品配置界面

14.8 用控制器来虚拟其他磁盘阵列

现在存储行业内常说的“存储虚拟化”，或者“虚拟化”，一般情况下都是指把多个相同或者不同厂商的磁盘阵列整合之后再分配的虚拟化方式。其实这种虚拟化只是冰山一角，而且纯技术架构设计方面的含量也不是很高，这么说并不是我在夸口，而是事实。实现这种虚拟化存储控制器的关键难点在于如何与后端不同厂商的设备之间兼容，包括SCSI Reservation的兼容等，以及如何发挥出后端存储设备的性能。所有不同厂商的设备虽然其内部设计等都不同，但是它们绝对需要相同的是前端的接口、包括物理接口、数据接口和逻辑接口。比如，它们至少都支持主机端Linux系统，包括支持Linux下的文件系统起始地址的对齐等。如果用一台Linux服务器来连接不同厂商的设备，那么这台Linux服务器就可以识别到所有的LUN并对这些LUN进行IO操作，如果在这台Linux服务器上实现某种软件层次，将所有这些LUN当作物理磁盘，在其上再次划分LUN，然后将这些LUN通过前端接口映射出去，那么这不就是一场虚拟存储控制器了么？这种虚拟化技术甚至可以通过DIY来实现，只要有合适的软件层，其他都不是问题。

IT系统内的最高深莫测和最复杂的虚拟化技术其实都在于主机端，即诸如Vmware等虚拟机软件产品及其附加功能，比如Vmotion等；以及诸如IBM等厂商的小型机上的硬件虚拟化、分区、动态迁移等功能；还有虚拟化的最高境界，整合了几乎所有虚拟化技术的云计算、云存储、云服务。

本节仅简要介绍一下目前市场上的一种虚拟存储控制器，其细节就不作过多介绍了，因为其架构并不复杂。

IBM公司的SVC产品

SVC的全称为SAN Volume Controller。图14-37所示为SVC的逻辑架构图。

图14-37 SVC逻辑架构图

整个系统架构可以分为4个层次。第1层是后端的磁盘阵列对外映射的LUN；第2层则是在SVC控制器内将这些LUN封装为Mdisk对象，每个LUN都为Mdisk，即Managed Disk，叫法并不重要，只需要知道Mdisk是后端的LUN即可，将多个Mdisk放入一个MD Group内，这个MD Group相当于一个Raid0模式的Raid组，一个系统内可以有多个MD Group；第3层，SVC在Raid组内划分LUN，生成的LUN称为Vdisk，即Virtual Disk，Vdisk相当于SVC自己的LUN；第4层，SVC将这些自己生成的LUN映射给前端主机使用。

SVC只能通过FC交换机来连接后端存储设备。前端主机则可以通过FC交换机（使用FC协议访问）或者以太网交换机（使用ISCSI协议访问）来访问SVC。

为了避免单点故障，每两台SVC控制器（node）可以组成一个IO Group，每个Vdisk同一时刻只能由一个node来掌管，当这个node故障之后，与这个node同处一个IO Group的另一个node接管之前故障node所管理的所有Vdisk。当IO Group中的两个node都故障时，其他IO Group的node会接管故障的node。目前一个SVC Cluster最多可以有4个IO Group（即8个node）。Cluster内的所有node必须都可以访问后端对应的存储系统的LUN。图14-38所示为一个8node的SVC Cluster。

图14-38 SVC集群示意图

这种存储虚拟化模式有诸多好处。从底层方面来看，多个后端LUN可以组成Raid0而不用担心物理磁盘损坏问题，因为后端存储设备会处理这一切；Raid0是性能最好的Raid模式而且还不用rebuild，至少能够将后端的性能影响屏蔽掉一部分；Raid0做成之后不需要初始化过程，LUN立即可用，方便快捷。从高层方面来看，这种虚拟化网关设备可以实现。

注意： 规划时需要注意的一个最大问题是：一定要清楚知晓SVC前端的LUN到底实际占用了多少块硬盘，是否有多个LUN共享同一组物理硬盘，前端主机的要求是什么，是否需要分布在同一组物理硬盘上的多个LUN进行并发的连续IO操作，这个问题具体可参考本书第19章。

图14-39所示为两个SVC控制器前视图。

图14-39 SVC控制器前视图

除了SVC这种架构的虚拟化网关之外，有些产品设计则是将虚拟化网关的功能集成到某个交换机中，比如F5公司的某系列产品就是将虚拟化功能集成到了以太网交换机中，或者说把以太网交换机集成到服务器硬件中，都可以。它可以针对后端NAS设备作虚拟化整合和再分配、数据迁移等，这个虚拟化网关将后端所有NAS设备的共享输出目录作为一个大的存储空间，将这个空间重新规划和分配，然后提供新的Export目录给主机客户端使用。

14.9 飞康NSS存储虚拟化系统

一直被效仿，从未被超越——这句话用在飞康身上很合适。飞康是存储软件领域的领军厂商，专注于存储虚拟化、容灾、CDP与数据备份领域。飞康在业界的地位取决于其两大优势产品和多项独特技术，其一是其IO级粒度回滚的CDP技术，其二便是其多功能融合的一体化存储虚拟化产品和诸如MicroScan等技术。目前飞康刚刚推出了IPStor Gen2 Plarform平台，其中包含了NSS、CDP、VTL三大产品系列，涵盖了企业数据保护备份、容灾、虚拟化全领域。

飞康主要面向企业级关键应用提供容灾、数据保护和虚拟化方案及产品，目前在全球有几千个用户的部署案例。这家以技术为核心的厂商，经过十余年的磨练，仍然保持领先，这一点难能可贵。

提示： 存储软件厂商不多，做出名堂的就更少了。关于飞康的IO级CDP回放技术，我将在第16章中详细介绍和分析，本节会着重分析飞康的虚拟化平台中的一些特色技术。

飞康NSS全称为Network Storage Server，它的本质是一台融合了高级数据保护、数据迁移和容灾功能的存储虚拟化网关，而存储虚拟化网关的本质，就是一款存储系统控制器，只不过它没有太多的后端接口来接入大量的磁盘扩展柜罢了，它的存储空间需要借用自其他存储系统，尤其是那些四肢发达、头脑简单的存储系统，这些系统自身往往功能和可靠性方面不是很出色，而此时如果加上NSS存储虚拟化网关的配合，则可以形成更强大的存储系统。

如图14-40所示为飞康NSS网关型管理器前视图，其硬件仅为标准

x86服务器，但是我们先不要以貌取人。那些定制化的SAN存储控制器，其表象上看似个性十足，但是要论硬件性能，不见得赶得上标准服务器，尤其是那些中低端产品。其次，存储系统的核心在于软件，目前的存储系统里几乎没有硬件加速逻辑了，因为这样做成本太高，而且缺乏灵活性。所以各家的可靠性、功能、性能，完全取决于软件的功能和优化程度。如图14-41及图14-42所示为NSS的高级技术和功能一览。本节会对这些技术做一一介绍和分析，我将飞康NSS中所包含的技术划分为三大类，分别为存储虚拟化相关技术和产品、数据保护相关技术和产品、异构迁移和容灾相关技术和产品。

图14-40 NSS网关型管理器前视图

图14-41 NSS网关型管理器特色技术一览

图14-42 NSS网关型管理器高级软件功能一览

14.9.1 存储虚拟化相关技术

对于一款虚拟化产品，广泛的兼容性是其立足之本。虚拟化本身没有太多意义，无非就是借用其他的LUN，然后形成自己的LUN，如果一款产品仅能做到这一点，那么LVM、Windows下的动态磁盘，都可以和独立虚拟化产品并驾齐驱了。一款虚拟化产品其虚拟化别人的LUN的目的是为了提供更多、更强的高价值服务。下面我们就详细介绍飞康IPStor平台所提供的增值服务。

SafeCache

SafeCache是飞康IPStor平台提供的数据加速访问技术。利用一些高性能存储介质（比如SSD、NVRAM等），这个技术可以将主机下发的

写IO数据像日志一样连续写入高速介质，快速响应主机IO，然后在后台异步地将这些数据刷到主存储空间中。对于SafeCache，系统需要维护一张映射表来追踪那些被缓存的块，一旦命中该映射表，则从SafeCache中读出数据。

如图14-43所示为SafeCache配置步骤，首先需要选择一个适当大小的存储空间作为Cache空间，可以手动选择或者交给系统自动选择。然后可以手动配置缓存刷盘策略，比如根据缓存高水位线、IO静默时间以及刷盘的力度。还可以感知那些写命令中的IO，降低重复刷盘的几率。

图14-43 SafeCache配置步骤1

HotZone

HotZone是IPStor平台下的另一个数据访问加速功能，其与SafeCache的区别是：HotZone是一种主动的数据访问速度优化技术，而前者则是一种被动方式的优化。HotZone的基本原理是将源卷划分为多个“zone”，然后对每个zone统计访问频繁程度，最后将那些热点zone中的数据缓存到高速存储介质中以加速读访问。

图14-44所示为HotZone的配置参数，可以选择缓存模式，比如是根据频繁程度来将那些热数据从低速介质中迁移到高速介质中，还是作为预读缓存仅仅加速那些连续大块IO读场景。具体参数可以配置预读启动临界点（连续地址IO被监听到多少次）、每次预读的容量、每次预读的IO Size、临界点时效周期。不得不承认这些参数非常专业。

图14-44 HotZone配置步骤1

图14-45所示为HotZone的其他配置参数，包括手动或者自动来选择充当缓存介质的存储空间，以及每个zone的容量，最小容量为64KB（其实这么小的粒度下，zone已经不适合描述了，Block更合适）。还可以配置每个zone在高速缓存中被缓存的最小时间。

图14-45 HotZone配置步骤2

图14-46所示为HotZone其他配置参数，包括访问频度统计模式，比如只读算、只写算，还是读写一起都算一次频度+1。右下图是一种可视化数据展现，图形化地展示每个zone的访问频繁程度，这就是一种可视化存储智能。

图14-46 HotZone配置步骤3

说明：可能是之前搞产品设计时遭遇对牛弹琴的原因，看到类似的Idea在飞康产品中出现，心情很复杂。

现在不少厂商推出了各种形态的固态存储产品，从简单的2.5寸SSD到用内存堆成的固态存储系统，由于价格昂贵，迫切需要一种合适的应用场景，而飞康NSS虚拟化设备恰恰提供了这种原生支持，将闪存阵列虚拟化之后，用在SafeCache/HotZone方案下，是目前这些固态存储系统性价比最高的归宿了。

Zero Memory Copy

这是一个很独特的I/O引擎算法设计。数据在通过NSS网关的时候，直接写入后端的存储，而不依赖于网关服务器的内存进行缓存或运算处

理。这样既提高了数据安全性，同时还保证了性能。一般来讲虚拟化网关产品都会在内部使用较大容量的RAM作为数据缓存。飞康认为使用这种数据缓存是杯水车薪并且多此一举。虚拟化设备后端本身已经挂接了拥有较大容量缓存的存储控制器，或者挂接各种新一代的闪存阵列产品作为全局缓存（如作为前述Safe Cache和Hot Zone的介质），此时在其前端再加上一点点缓存，还不如不加，因为这点缓存相比后端的缓存容量是小巫见大巫。更何况，仅仅由于增加的这一点缓存，由于实质上对生产数据产生驻留还将导致数据安全的隐患。飞康对于企业级数据安全的理解是深刻的，存储虚拟化网关应该在即便整体全部意外故障或失效的时候也不会对原生产数据的安全带来任何影响。

另外加了缓存需要付出三个代价：第一，RAM本身需要成本并需要电池维护，这不但增加了采购成本也提升了维护成本；第二，RAM的易失性要求双机的RAM实时向对方同步，（各自实际使用二分之一空间），任何写入缓存的I/O只有在缓存确认向对方同步后才算写入成功，这样，横向同步的设计和性能可能成为网关的写入性能瓶颈；第三，这部分RAM势必要不断地刷盘，也就是刷到后端挂接的存储系统中，批量刷盘会对后端链路瞬间爆发式占用，一定程度下还会进一步影响性能。飞康采用零缓存拷贝方式，降低了功耗、复杂度和成本，虚拟化控制器只作为通道，不缓存数据。在保证性能的前提下，保证了自身双机失效极端情况下的数据安全性和稳定性，并且降低了成本。

Alternate-READ/Smart-READ

Raid1是很简单的技术，两个物理磁盘，或者两个LUN/卷之间做成镜像关系，但是很少有产品把优化做到极致，比如很多人认为Raid1的两个盘/卷之间一定会做读负载均衡，也就是读IO会轮询发给这两个盘

去执行，但是事实上，因为做到这一点需要额外的开发，并不仅仅是做个负载分发器简单地分派IO就完了，还需要处理数据一致性，比如在异步镜像模式下，如果源卷的某些数据尚未同步到镜像卷，而此时分发器就需要先判断目标IO是否可以先发送到镜像卷执行，它需要先搜查元数据（比如bitmap之类），这是需要开发和验证测试工作量的，一些头脑简单、四肢发达的产品基本是不会考虑做这些优化的，因为很多用户本身并不专业，不会注意到或者理解这些高技术含量的东西，还不如用低价+高硬件配置来解决问题。飞康显然走的是另一条路线，以智取胜，其Alternate-READ技术，可以实现镜像卷的读负载均衡。

另外，数据复制、快照等等功能，都需要从源卷来读取数据，这显然会影响主机侧的IO性能，飞康的Smart-READ技术，可以让这些阵列内自身发起的读IO被重定向到镜像卷去执行，这样就可以分担源卷的读IO压力了。

提示： 将技术做到极致是我研究飞康技术以来的第一印象。

图14-47 选择不同的镜像读方式

如图14-47所示，在创建存储镜像卷的时候可以手动选择三种镜像读模式，分比为Smart-READ模式、Alternate模式和Exclude Mirror模式，前两者上文已经介绍过了，最后一种模式就是让系统永远不从镜像卷读数据。

High Availability Cluster

NSS支持集群互备模式，双控制器各自分工，各自对后端的一部分LUN进行虚拟化管理，平时各管各的，但是在一个控制器宕机之后，另一个控制器便接管起对方的LUN继续做虚拟化管理。

14.9.2 数据保护相关技术和产品

Snapshot/ Data Journaling/CDP

每个卷最大支持1000个快照。系统还支持针对各种主流应用的 Application Aware Snapshot Agent代理程序，可以实现主流应用系统的数据一致性快照。支持一致性组，一致性组中的多个卷之间可以保证时序一致性。支持TimeView现场预览模式。

提示： 关于飞康快照和CDP的详细描述可以参考本书第16章相关内容。

Recovery Agent

Recovery Agent是飞康针对各种Windows下的主流应用（比如Exchange、SQL Server、Lotus Notes、Volume Shadow Copy Service）提供的图形化恢复工具，其与飞康NSS/CDP设备、Snapshot Agent以及应用系统三方之间紧密配合，从而可以让管理员在无需介入复杂的应用系统的情况下，很容易地实现从应用视角切入的数据恢复。

14.9.3 异构迁移和容灾相关技术和产品

SED

Service Enabled Device，中文不好翻译，但是这个功能是至关重要的。作为一款专业的存储虚拟化设备，其人品应该是优秀的，所谓人品优秀，就是不会只知道强制“霸占”别人的资源，有时候还需要懂得借花献佛和激流勇退。

怎么理解？虚拟化产品一般都是将从其他存储系统处借来的LUN当做物理磁盘，然后在其上再做一层虚拟化，虚拟出一堆LUN来。但是有的时候，用户不需要二次虚拟化，而要求之前的LUN保持不变，其上数据也不被损坏，甚至连这个LUN的厂商、序列号等都保持不变，但是同时需要针对这个LUN实现快照、复制、CDP、镜像等高级功能。另外，在数据迁移场景下，虚拟化设备必须完全模拟原来的LUN的所有属性，因为只有这样上层应用才能保证不出任何兼容性问题，同时把该LUN中的数据透明迁移到另外的存储设备中去，然后做切换，而且必须保证OS识别到这个LUN之后不会认为它是个新LUN从而导致盘符错乱，盘符顺序必须保证一致，这样才是对应用完全透明，所以要求该LUN的序列号、文件系统格式等完全一致，也就是该LUN中的每个扇区都不会被虚拟化设备私自去修改，这就是借花献佛，不将该LUN写入自己的标签，据为己有，这有点类似于带外虚拟化。虚拟化设备会将原本应该写入原LUN的标签等元数据，单独记录在其他地方。

所谓激流勇退，是指一旦当虚拟化设备出现什么问题，那么此时应用的IO会全部中断，业务停止，为了给这种情况留有后路，虚拟化设备就必须将原来的LUN透明地传给主机访问，这样的话，当虚拟化设备出问题之后，可以拿掉虚拟化设备，重新修改LUN映射，恢复之前的拓扑，让主机直接连接之前的存储系统，此时主机依然会识别到这些LUN，挂起之后盘符也不会变化。但是如果虚拟化设备不透传原来的LUN，而在LUN上写入了自己的管理标签，并且对这些LUN做了二次虚拟化之后，那么一旦虚拟化设备出了问题，将不会有任何设备能够识别这些LUN，因为这些LUN里的格式已经不是物理硬盘的普通分区格式了，甚至连MBR/GPT格式都不是，而是虚拟化设备厂商私有的格式，所以OS根本挂不起来这些LUN，只会认到“新磁盘”，会提示“需要格式化”，所以此时除非找到同样厂商的同样设备来恢复业务，否则业务就

要一直停机。所以专业的虚拟化设备都会提供这种LUN透传模式。这种模式俗称“逃生模式”。

如图14-48所示为飞康NSS设备所提供的两种模式。可以看到，在SED模式下，系统还可以让用户选择是否连原LUN的厂商信息等也完全保留（Preserve physical device inquiry string），比如原来的LUN是厂商A存储系统的，OS设备管理器里是可以识别到厂商A的，其底层其实是使用了SCSI Inquiry LUN命令来得到这些信息的，虚拟化设备只要用原LUN的这些信息来响应主机的SCSI Inquiry LUN命令即可。

图14-48 SED模式和普通虚拟化模式

SED模式的LUN可以被快照（必须为CoFW模式）、镜像、复制，但是不能被Enable Thin Provision，因为Thin Provision会改变原LUN中原有的数据布局，导致不可能透传给主机使用。

提示： 数据迁移为何有难度？就是因为要尽量缩短停机时间。假设可以无限停机，那么此时完全可以用dd来盘对盘复制实现数据迁移，迁移前和迁移后，两个盘上的每个扇区都是一样的。但问题是停机不能太久，否则dd根本搞不定，dd不支持在源LUN不停地接受写IO的同时，还能把这些新的更新数据同步迁移到对端去。那么是否可以使用LVM或者软Raid来将两个LV做镜像，同步之后，拿掉源存储系统呢？完全可以，但是要求应用主机必须使用LVM或者软Raid，另外有些应用或者主机根本不使用LVM，就直接用/dev/sda，此时LVM/软Raid没辙。那么，针对/dev/sda，是否可以先用LVM将两个/dev/sdxx设备做镜像，然后再让LVM退出，还原成/dev/sda的原有格式？不可能，要将某个/dev/sdx纳入LVM的管理，那么LVM便

会向其中写入自己的管理标签和元数据，这就已经破坏了该LUN中原有的数据了，就更别提还原了。所以，数据迁移场景，必须使用类似飞康SED这种模式。

MicroScan

远程数据复制，本质上没什么技术含量，无非就是将数据复制到远程，这就像从网站上下载一个文件到本地一样。虽然企业的数据容灾链路如我们的Internet一样也是包月的，但却是专用的、不与其他人共享的，价格高、带宽低、时延大。企业每天都会产生大量数据，在这种低带宽、高时延链路下进行数据复制，就得用一些特殊的技术。飞康MicroScan是其专利技术，能够节约大量数据复制流量，让企业可以用更低带宽的链路达到其他数据复制产品同样的效果。

提示： 首先，多数人不知道的一个事实是，数据复制时产生有大量冗余，源和目的端本来就相同的数据，被复制到远端，究其原因，是因为底层磁盘是按照512Byte扇区作为最小IO粒度，而文件系统一般使用4KB作为最小管理粒度，上层Page Cache同样也是4KB。如果应用要更新某个字节，那么文件系统也会将这几个字节所落入的4KB数据块读入Page Cache，然后更新这几个字节，再写入整个4KB块，这就是读写惩罚，本书第19章会对这个现象做详细分析。这样的话，假设应用只更新了一个字节，那么文件系统白白读出了4096字节，又白白写入了4095字节，而底层的数据复制引擎也就必须将这4095字节冗余数据再次复制到远端，虽然远端的这4095字节与本地的4095字节一模一样。

MicroScan技术可以降低上述冗余粒度到一个扇区，只要某个扇区

发生变化，那么不管上层是按照什么粒度来管理存储空间的，也不管上层的写惩罚有多大，底层会发现那些发生了变化的扇区，从而只复制这些扇区到远端。

提示： 这项技术的底层原理，有技术感觉的读者一眼就可以判断出，一定是利用了与重删一样的技术，就是算Hash。没错，但是重删的目的是把数据压实以缩小体积，而MicroScan的目的是发现变化，节省复制的带宽。所以我在此推测一下MicroScan技术的底层原理。首先，在初次复制之前，复制引擎一定会对源LUN整盘的每个扇区都计算一个hash值然后保存起来，按照64bit hash值来算，1TB的LUN就需要16GB的hash存储空间/hash库，这些hash值必须按照扇区顺序来排序以加快IO速度，hash库只会被保存在源端。

当数据复制还没有开始之前，每当源卷发生一笔写IO，系统便会以扇区为单位算好新数据的hash，然后异步更新到hash库里，这样，hash库时刻处于最新状态。当数据复制开始之后，第一笔针对源卷的写IO，系统同样会以扇区为单位计算其hash值，然后从hash库中对应偏移处取出针对这笔IO目标地址处所存放的原来数据的hash值，进行比对，发现哪些扇区的hash值发生了变化，然后便向一个链表中追加提交该发生变化的扇区，等待复制（须先搜索该链表，查看是否之前已经有针对该扇区的复制任务，如果有便删除之，以最新的为准，这样可以保证IO复制的时序一致性）。由于hash库是完全按照扇区地址排序的，所以不存在“搜索”这一步，系统直接套用偏移量算式代入一步即可得出目标hash的位置，然后直接发起IO读出hash数据比对即可，所以虽然元数量非常大，但是根本无须载入内存。

如图14-49所示为MicroScan技术能够节省的带宽比例。

图14-49 MicroScan技术效果示意图

RecoverTrac

容灾最难的是什么？不是数据复制，不是配置安装，而是前期规划和平时的运维管理和演练，当真有灾难发生的时候，切换就是点一下鼠标，一锤子买卖的事情。有些双机HA产品，它们规划、管理非常简单，因为往往只有两台机器和一套应用。但是如果是从双机到多机（比如几十台机器），再到虚拟机（比如几百台机器），应用也从一套单一应用到多套应用，而且应用之间相互关联依赖，对于这样一个系统的容灾，其管理复杂度就很高了，需要一个集中的容灾管理系统来帮助管理员管理容灾体系。

Stretched Cluster

与众多厂商的Stretched Cluster解决方案架构一样，在用户的两个数据中心分别部署一套存储和一台NSS存储虚拟化网关，两个数据中心NSS存储虚拟化系统互为备份，两套存储系统可以是不同品牌，通过飞康NSS Stretched Cluster虚拟化网关互为镜像保护；两台NSS存储虚拟化网关都可以对外提供服务，把磁盘卷分配给各自站点的主机使用，也就是实现了两个站点负载的相互分担。

在两个数据中心均工作的情况下，当某一数据中心NSS虚拟化网关发生故障时，另一数据中心的NSS虚拟化网关可以自动接管故障NSS上运行的任务，接管过程大约需要20秒左右，业务系统运行不会中断，保证了业务系统的连续运行。当某一数据中心存储系统故障时，作为镜像的另一数据中心存储自动顶替生产，上层运行的应用系统和数据库系统

对存储的故障无任何感知，业务系统运行不会受到任何影响，从而保证了业务的连续性。当发生主机系统故障时，可以由主机层的HA环境自动切换应用系统和数据库系统程序到另一个数据中心的主机上运行。

为了防止脑裂现象，飞康NSS Stretched Cluster引入了一个第三方仲裁机制—Tiebreaker仲裁。它与两个NSS存储虚拟化网关的电源控制模块进行通信，支持IPMI和HP iLO。Tiebreaker服务器通过网络监控Cluster节点的电源控制模块，当Cluster虚拟化网关之间无法判断对方状态时，Tiebreaker会对收不到响应的虚拟化网关进行重置。阻止两个节点都向存储尝试写入的情况发生，从而避免数据不一致现象。

第15章 众志成城——存储集群

- 分布式
- 集群
- 高可用性集群
- 负载均衡集群
- 高性能集群

随着应用程序对服务器和存储系统的要求越来越高，对于传统设备来说，比如PC、PC服务器、小型机服务器等，单台设备有时已经不能满足需求了。此时虽然可以使用大型机，在单一设备上提供更高的性能，但是大型机的物质成本和维护成本是高不可攀的，而且大型机也不见得适合所有应用。怎么办呢？众人拾柴火焰高，人们想出了一种办法来应对日益扩张的应用程序需求，就是用多台设备联合起来对外提供服务，这就是集群。

主机可以形成集群，存储设备一样可以形成集群。目前中高端存储设备其自身就具备双控制器。不但如此，有一些NAS设备还可以在众多台独立设备之间形成集群，并且实现了单一名称空间，即用户访问目录路径像访问一台机器一样，而实际上，可能是由集群中不同的节点来提供服务。

15.1 集群概述

用多个节点来代替一个节点完成任务，毫无疑问是为了提高处理能力。其次，集群还可以做到高可用性，即一旦某个节点发生故障，不能再继续参与计算，那么集群中的其他节点可以立即接替故障节点的工作。

15.1.1 高可用性集群（HAC）

在HA集群中，节点分为活动节点和备份节点。活动节点就是正在执行任务的节点；备份节点是活动节点的备份。一旦活动节点发生故障，则备份节点立即接替活动节点来执行任务。高可用性集群的实现是基于资源切换的。所谓“资源”是指HA集群中某个节点发生故障之后，备份节点所要接管的任何东西的一个抽象的词汇。比如，在某个节点发生故障之后，其对应的备份节点，需要接管故障节点上的IP地址、主机名、磁盘卷、应用程序的上下文等，这样才能将对客户端造成的影响缩减到最小。这些被接管的实体，便被称为“资源”。资源的监控和接管，依靠于HA软件。目前存在多种HA软件。每种操作系统几乎都自带HA软件，它的作用就是监控对方节点的状态，一旦侦测到对方的任何故障，那么便会强行将所有资源占为己有并向客户端继续提供服务。

15.1.2 负载均衡集群（LBC）

在负载均衡集群中，集群中的所有节点都参与工作，每个节点的地位相同，接受的工作量按照某种策略，由一个单独的节点作为调度来向其他所有参与运算的节点分配，或者由所有参与运算的节点之间通过网络通信来协商分配。分配策略如轮流分配、随机分配、最小压力分配

等。

15.1.3 高性能集群（HPC）

高性能集群，又称科学计算集群。这种集群其实与LBC集群的本质是相同的。只不过其专用于科学计算，即超大运算量的系统，比如地质勘探、气象预测、分子筛选、仿生模拟、蛋白质构型、分子药物分析、人工智能等。这些运算要么逻辑复杂，要么需要大量穷举，会耗费大量的CPU和内存资源。有些需要几天、几个月甚至半年才能执行完毕。此时，增加整个系统的CPU总核心数，可以成倍地缩短执行时间。

提示： 记得笔者在大学做作业设计的时候，有个同学的课题就是分布式计算，这课题也简单，就是第一天将任务执行上，一个月之后结果出来了，写论文、答辩。那时候的计算机上用的都是Intel奔腾4的CPU，倘若用现在的酷睿多核CPU，我想只需要十几天便可以出结果。

HPC集群中，为了增加整个系统的CPU核心数，一般引入十几台或者几十台、几百台计算机，其中每台计算机又可以有多个物理CPU，每个CPU又可以有多个核心。这样整个系统的CPU核心数会相当可观。那么如何利用这么多的CPU呢？如何将任务平均分配到每个CPU核心上呢？

Windows 2000以后的Windows系统，操作系统默认便自动支持同一台计算机内的多个CPU或者多个CPU核心，操作系统自动将多个线程平摊到多个CPU核心上运行。但是对于不处于同一台计算机内的CPU来说，任务将要怎么分配到其他节点上呢？当然是通过网络了。为了方便编程出现了很多API，为程序员屏蔽掉多CPU所带来的编程复杂度，程

程序员只要按照这些API规范来编写代码，底层便会自动将运算任务分派到网络上的其他运算节点上。节点接收到任务数据之后，再由节点操作系统自行将这块任务数据分派到节点的多个CPU核心上。MPI便是一个目前广泛应用的HPC系统API。

15.2 集群的适用范围

集群可以实现在系统路径的任何点上。

硬件上：CPU、内存、显卡、显示终端、以太网卡、计算机本身、以太网及IP网络设备、FC卡、FC网络交换设备、磁盘阵列控制器本身、磁盘阵列控制器内部的各个组件、磁盘本身、磁盘内部的多片盘片和多个磁头。

软件上：应用程序、文件系统、卷管理系统。

什么时候需要实现集群呢？

当某个系统的处理能力不能满足性能要求的时候，可考虑使用负载均衡集群或者高性能集群；当追求系统的高可用性时，即希望某处故障不会影响整个系统的可用性的时候，使用高可用性集群；当需要运算的数据量很大，运算周期很长的时候，可考虑实施高性能集群。

目前，各大知名网站一般都采用负载均衡集群来均衡TCP连接请求。由于这些网站每天的访问量很大，同时产生的TCP连接请求也很多，所以如果只用一台计算机来接受这些请求，根本满足不了性能，甚至会造成这台机器资源耗尽而死机。基于Linux系统的LVS，是由国人主持研发的一种TCP负载均衡软件，被广泛用于TCP连接压力很大的系统下。LVS可以基于很多策略来将前端的请求分摊到后端的多台计算机上。其本质就是一个基于策略的TCP包转发引擎。

对于比较重视IT建设的企业、重要的应用系统，都可实施HA集群来追求高可用性，从而避免故障造成的生产停顿。各大科研院所、气

象、石油勘探等机构，由于其需要很大的运算量和运算周期，一般都有HPC集群。

15.3 系统路径上的集群各论

15.3.1 硬件层面的集群

图15-1中箭头指向的部件都可以被集群化。

图15-1 系统路径上可实现集群的各处

（1）CPU的集群。应用在多CPU的计算机系统，比如对称多处理器系统，多个CPU之间共享物理内存的共同协作。目前的服务器以及小型机系统大多为这种结构。

（2）内存的集群。多条物理内存组成更大容量的空间，并且通过比如双通道（相当于磁盘系统中的条带化RAID 0）等技术，提高性能。

（3）以太网卡的集群。目前有多种方式来实现以太网卡的集群。将主机上的多块以太网卡绑定，向上层提供一块虚拟网卡，底层则可以通过ARP轮询负载均衡方式，或者802.3ad方式等向外提供负载均衡，或者HA方式的多路径访问。

（4）以太网及IP网络设备的集群。在以太网交换机和IP路由器上，多台设备之间协作转发网络数据包（帧），诸如Cisco、华为等厂商都已经实现了负载均衡以及HA方式的集群。

（5）显卡的集群。显卡集群是最近出现的技术。NVIDIA以及AMD公司都有对应的解决方案。将插在总线上的多块显卡通过特殊连线连接起来，实现对大型3D数据渲染的负载均衡，性能得到很大提升。

（6）显示器集群。比如电视墙等。但是这个严格来说并不算作集群。

（7）FC卡的集群。通过与主机上的多路径软件配合，多块FC卡之间可以实现流量的负载均衡和HA。或者通过FC网络中的ISL链路负载均衡、HA方式实现流量分摊。

（8）FC网络设备的集群。目前来说，FC网络设备并没有像以太网以及IP网络设备那样实现负载均衡以及HA。但是很多网络存储系统中，一般都部署多台FC交换机以避免单点故障，但是这个环境中的FC交换机本身并没有集群智能，所有集群逻辑都运行在FC节点上。

（9）磁盘阵列控制器集群。目前几乎中高端的磁盘阵列的控制器都为双控架构，两个控制器之间可以为HA关系，或者为负载均衡关系。

（10）磁盘的集群。典型的磁盘集群就是RAID系统，7种RAID（磁盘集群）方式，这里就不多描述了。其次磁盘内部的多块盘片，多个磁头之间也组成了集群，但这并不能算作集群，因为同一时刻只能有一个磁头在读写。

15.3.2 软件层面的集群

软件层面的集群如图15-2所示。

图15-2 软件层面的集群

1. 应用程序的集群

一个应用程序可以同时启动多个实例（进程），共同完成工作。应用程序的不同实例可以运行在同一台机器上，也可以运行在不同的机器上，之间通过网络交互协商信息。

2. 文件系统的集群

文件系统的集群是一门比较独立的课题。可以实现集群功能的文件系统称为集群文件系统。比如NFS、CIFS等网络文件系统，就是最简单的集群文件系统。

集群文件系统的出现主要是为了解决三个问题：容量、性能、共享。

（1）容量问题。集群文件系统有一类又被称为分布式文件系统。即某个全局目录下的存储空间，实际上是分布在集群中的各个节点上的。分布式文件系统将每个节点上的可用空间进行虚拟的整合，形成一个虚拟目录，并根据多种策略来判断数据的流向，从而将写入这个目录的数据对应成实际存储空间的写入。这样便可以做到集群中的整合存储，充分利用集群的资源优势。

（2）性能问题。用多个节点共同协作来获取高性能，这在文件系统层次依然成立。集群文件系统使得每个节点不必连接昂贵的磁盘阵列，就可以获得较高的文件IO性能。在分布式文件系统的虚拟整合目录的做法之上，又采取了类似磁盘条带RAID 0的处理方式，依据各种负载均衡策略，将每次IO写入的数据，分摊到所有节点上，节点获得的性能越多提升就越大。但这只是理论情况，实际使用起来集群文件系统并不是一个容易实施的系统，实施之后想要获得高性能，必须经过长时间的优化调试过程。

(3) 共享访问。集群文件系统所解决的最后一个问题，也是最为重要的一个问题，就是多节点共同访问相同目录和相同文件的问题。集群文件系统对多个节点，同时读写相同的文件做了很周全的考虑，能保证所有节点都能读到一致性的数据，并且利用分布式锁机制保证在允许的性能下，节点之间不会发生写冲突。

常见的集群文件系统有PVFS、PVFS2、Lustre、GFS、GPFS、DFS、SANFS、SANergy等，这里就不做过多介绍了。

3. 卷管理系统的集群

本机的卷可以与本机卷或者远程计算机上的卷进行镜像等协同操作，形成集群。

15.4 实例：Microsoft MSCS软件实现应用集群

Windows Server 2003集群要求每台服务器上至少有两块以太网络适配器，一块作为公用网络适配器（连接外部网络），一块作为专用网络适配器（用于心跳检测）。集群中的所有节点必须在同一个域中，一般双机环境中直接使用其中一台为主域控制器，另一台为备份域控制器。

15.4.1 在Microsoft Windows Server 2003上安装MSCS

使用“控制面板”的“添加/删除程序”工具，添加Windows组件，安装集群服务。

（1）在管理工具菜单中打开集群管理器，当弹出集群连接向导时，选择“创建新集群”，并单击“下一步”按钮继续，如图15-3所示。

图15-3 创建新集群

（2）输入集群的唯一NetBIOS名称（最多15个字符），单击“下一步”按钮，如图15-4所示。

图15-4 输入集群名称

（3）如果在本地登录一个不属于“具有本地管理特权的域账户”的账户，向导会提示用户指定一个账户，如图15-5所示。

图15-5 输入集群账户信息

（4）确认将要作为第一个节点创建集群的服务器的名称，如图15-6所示。

图15-6 输入节点名称

(5) 安装程序将分析节点，查找可能导致安装出现问题的软硬件问题。检查所有警告或错误信息。单击“详细信息”按钮可以了解有关每个警告或提示的详细信息，如图15-7所示。

图15-7 检查集群配置环境

(6) 输入唯一的集群IP地址（只能用于管理，不能用于客户端连接），如图15-8所示。

图15-8 输入集群IP地址

- 输入在安装时创建的集群服务账户的“用户名”和“密码”。
- 集群配置完成，单击“完成”按钮结束。
- 集群配置完成后，选择磁盘阵列上的一个LUN为仲裁盘。
- 完成节点1的配置后，在另一台机器上也安装集群服务，完成后打开集群管理器。
- 当弹出集群连接向导时，选择“加入现有的集群”，根据向导完成节点2的配置。

提示： 仲裁磁盘（Quorum Disk）用于存储集群配置数据库检查点，以及协助管理集群和维持一致性的日志文件。仲裁盘可以是一个逻辑分区，也可以是一个单独的磁盘。

15.4.2 配置心跳网络

(1) 启动“集群管理器”。

(2) 在左窗格中，单击“集群配置”，再单击“网络”，右击用于专

用网络（心跳检测专用）的适配器，从弹出的快捷菜单中选择“属性”命令。

（3）选中“仅用于内部集群通信（专用网络）”单选按钮，如图15-9所示。

图15-9 配置专用网络（1）

（4）单击“确定”按钮。

（5）右击用于公用网络的适配器，从弹出的快捷菜单中选择“属性”命令。

（6）选中“针对集群应用启用该网络”复选框，如图15-10所示。

图15-10 配置公用网络（2）

（7）选中“所有通信（混合网络）”单选按钮，然后单击“确定”按钮。

15.4.3 测试安装

在“安装”程序结束后，有几种验证集群服务安装的方法，具体如下。

- 集群管理器：如果仅完成了节点1的安装，启动“集群管理器”，然后尝试连接到集群。如果已安装了第二个节点2，可在任意一个节点上启动“集群管理器”，然后确认第二个集群显示在列表上。
- 查看启动服务：使用管理工具中的“服务”选项，确认集群服务已

显示在列表上并已启动。

- 事件日志：使用“事件查看器”检查系统日志中的ClusSvc条目。会看到有关确认集群服务已经顺利形成或加入一个集群的条目。
- 集群服务注册表项：确认集群服务安装程序将正确的项写入注册表。可以在HKEY_LOCAL_MACHINE\Cluster下找到许多注册表设置。

选择“开始”→“运行”菜单命令，然后在弹出的对话框中，输入“虚拟服务”名称。确认可以连接并看到资源。

15.4.4 测试故障转移

验证资源将执行故障转移。

选择“开始”→“程序”→“管理工具”菜单命令，然后单击“集群管理器”，如图15-11所示。

图15-11 集群管理器主界面

右击“磁盘组1”组，然后单击“移动组”。该组及其所有资源将转移到另一个节点。稍后，磁盘F：、G：将在第二个节点上实现联机。在窗口中观察该转移。退出“集群管理器”。

15.5 实例：SQL Server集群安装配置

上面我们已经设置好了MSCS集群基础平台，下面介绍如何在这个平台上安装SQL Server数据库。SQL Server 2000的集群安装配置已经直接集成到了SQL Server 2000的数据库安装向导中，能够自动识别到Windows Server 2003上的集群系统并启用数据库虚拟服务器选项，实现SQL Server 2000集群虚拟服务器在两台服务器上的自动安装配置。安装完成后，需安装SQL 2000 SP3补丁包。

确保SQL Server 2000集群在两台服务器上的自动安装配置，两台服务器MS-Clus-01A与MS-Clus-01B，以及共享磁盘柜都须处于开机在线状态。

15.5.1 安装SQL Server

(1) 在接管了SQL数据盘（磁盘Y：）的节点服务器MS-Clus-01A上，放入SQL Server 2000企业版安装光盘，启动SQL Server 2000的安装向导，如图15-12和图15-13所示。

图15-12 共享磁盘Y被MS-CLUS-01A节点
掌管

图15-13 安装SQL Server1

(2) 安装向导进入“计算机名”界面后，会自动识别到Windows Server 2003的集群系统，选择“虚拟服务器”选项，输入虚拟SQL Server名称“MS-Clus-SQL”，单击“下一步”按钮，如图15-14所示。

图15-14 输入虚拟SQL Server名称

(3) 在“故障转移集群”对话框中输入IP地址“192.0.0.4”，选用网络Public，单击“添加”按钮，使其添加到列表中，即这个IP地址属于公用网络。然后单击“下一步”按钮，如图15-15所示。

图15-15 配置虚拟服务器IP地址

(4) 在“集群磁盘选择”对话框中选择“组0”的共享磁盘“Y:”，然后单击“下一步”按钮，如图15-16所示。

图15-16 选择用于存放数据的磁盘

(5) 在“集群管理”对话框，确保MS-CLUS-01A与MS-CLUS-01B都在“已配置节点”列表中，然后单击“下一步”按钮，如图15-17所示。

图15-17 选择集群中要使用到的节点

(6) 在“远程信息”对话框，输入用户名、密码及域名，然后单击“下一步”按钮，如图15-18所示。

图15-18 输入账户信息

(7) 在“实例名”对话框，选中“默认”复选框，然后单击“下一步”按钮，如图15-19所示。

图15-19 实例名窗口

(8) 在“安装类型”对话框中选中“典型”单选按钮，由于前面磁盘选择了“组0”的“磁盘Y:”，“目的文件夹”的“数据文件”自动定位到Y:盘，而SQL程序文件则会自动安装到MS-Clus-01a与MS-Clus-01b的本地

图15-20 安装目的选择

盘相关目录下，单击“下一步”按钮，如图15-20所示。

（9）在“服务账户”对话框，选中“对每个服务使用同一账户”单选按钮，由于是集群配置，“使用本地系统账户”单选按钮为不可用。输入用户名、密码及域名，然后单击“下一步”按钮，如图15-21所示。

图15-21 输入账户信息

（10）在“身份验证模式”对话框中选中“混合模式”单选按钮，输入sa密码，然后单击“下一步”按钮，如图15-22所示。

图15-22 身份验证配置

（11）安装完成后，打开“集群管理器”，在“集群配置”下可看到“资源类型”中多了两个SQL Server的资源，这是因为SQL Server 2000企业版为Cluster-Aware的应用系统，安装配置时自动添加了支持Cluster的服务组件，如图15-23所示。

图15-23 新加入的资源（1）

（12）单击“组0”，可看到除原有的“磁盘Y：”外，新添了5个SQL资源，而且都已联机，说明SQL Server 2000集群安装配置完成，如图15-24所示。

图15-24 新加入的资源（2）

15.5.2 验证SQL数据库集群功能

（1）在一台客户机上安装SQL Server 2000的“企业管理器”与“查询分析器”，来测试验证数据库的FailOver功能。

(2) 打开“企业管理器”，注册数据库“192.0.0.4”，即虚拟数据库的IP或服务器名，如图15-25所示。

图15-25 连接到虚拟服务器

(3) 新建一个测试数据库MytestDB，如图15-26所示。

图15-26 创建新数据库“MytestDB”

(4) 在数据库MytestDB中新建表employee，并添加几条记录，如图15-27所示。

图15-27 新建表

(5) 打开“查询分析器”，连接到192.0.0.4的数据库MytestDB，检索表employee返回数据，如图15-28所示。

图15-28 employee返回数据

(6) 在“集群管理器”中移动数据库资源组“组0”（MS-Clus-01A → MS-Clus-01B），进行资源切换，如图15-29和图15-30所示。

图15-29 手动切换资源（1）

图15-30 手动切换资源（2）

(7) 移动“组0”过程中，在“查询分析器”中持续执行数据检索，刚开始“连接中断”检索不到数据，几十秒钟后又恢复正常，能顺利检索到数据，如图15-31和图15-32所示。

图15-31 服务中断 图15-32 服务恢复

测试验证表明，此配置方案可以实现数据库的FailOver，而且切换时间在30秒左右。

15.6 块级集群存储系统

存储系统一直以来都是以一个总控制器中心加上几串磁盘扩展柜的形式多存在的。直到最近几年，这种模式有被打破的趋势。

1. Scale-Up和Scale-Out

当一个存储系统使用一个或者两个冗余的控制器来对外提供服务时，如果其性能不能满足越来越苛刻的应用要求和负载，那么厂商会考虑在单个控制器中加入更多的CPU、升级更高的CPU主频、增加更多的内存以及扩充更多的IO总线以连接更多的IO扩展卡，以此来提高这个存储系统的性能，这种扩展方式称为Scale-Up。

而在一个独立计算机系统总线中增加更多的CPU和内存的做法，会越来越窄，随着CPU数量的增多，其所耗费的设计成本和硬件成本就会更陡峭地升高，而且，随着单系统内CPU数量的增加，其性能所得到的提升也会越来越趋近于0；其次，这种扩展方式也不利于长远发展，如果随着时间的推移，升级后的系统又变得不够用，那么不可能无限制地添加CPU和内存，Scale-Up的扩展方式等于是自寻死路。此时，就催生人们考虑是否可以将存储系统集群化，用多台CPU和内存以及IO扩展卡数量较少的独立的控制器，通过某种互联网络通道将所有控制器连接起来形成一个集群系统，这种扩展方式称为Scale-Out。

两种方式比起来，Scale-Out更加节约成本，而且在设计良好的情况下，可以增加更多的集群节点，系统性能随着节点数量的增加而近乎正比地增加。

2. 分久必合，合久必分；分中有合，合中有分，万物皆和合

与主机系统的集群化趋势一样，存储系统也步入了集群化。然而大规模主机集群真正广泛应用的是HPC领域，并未被一般企业所采用，其根本原因其实是因为对应的应用程序还并未跟上时代，仍处于非集群化开发模式下。虽然可以让非集群化应用运行于集群中某单个节点上，但是鉴于集群中的单节点性能反而可能不能满足需求的尴尬境地，所以大多数企业宁愿花费更多的钱使用性能强劲的PC Server甚至小型机来运行这些应用。只有一些高端的应用比如数据库等系统提供了集群方式的部署，比如Oracle RAC和DB2 PureScale等。而另一方面，企业又会被主机性能的浪费所困扰，从而部署虚拟机系统。所以，集群也不是，不集群也不是，这又催生了一种更加彻底的解决方案，即云系统，这个话题将在本书其他章节讨论。

对于集群存储系统，仔细体会一下，我们隐约可以洞察到其中有些奥妙之处。集群存储系统的数据分布有两种方式：

- 一是将多份整体数据每一份都分开存放于集群中的每个节点上；
- 二是将多份独立的数据，每一份放在一个独立节点上或者手动决定数据分割的份数以决定利用的节点数目。

对于前者的数据分布方式，在多个应用系统共同访问时，可能在特定情况下会影响系统整体性能，对于后者分布方式，每个应用系统只访问一个节点或者按照性能要求访问多个节点，对其他节点性能没有影响，这就相当于在一个传统非集群存储系统中，对每个应用建立独立的Raid Group一样的道理。

对于第二种集群数据分布方式，需要很多的人为介入，但是却可以

保证性能资源的平衡合理分配。任何存储系统的一个最大的问题就是在多主机多应用并发访问时如何保证系统的性能，特别是高带宽吞吐量的情况下（这个话题将在其他章节论述）。目前解决这种并发情况下性能大幅降低的一个办法就是隔离相关资源。在集群存储系统中，这个问题依然存在，所以依然要使用资源隔离的方法。所以，整个系统仿佛又回到了DAS架构，每台主机都使用自己专供的存储系统，只不过这些DAS存储孤岛被集中管理了起来而已。这种状态有一丝诡异，显示了这种事物处于一种若即若离的不定状态，下一步的发展，非收敛即分散，而我们调查一下这个事物在进入这种状态之前是什么状态呢？是收敛状态，即数据集中存储和访问，那么我们可以推断这个事物从收敛状态走向松耦合迷离状态，当前就是这样一种状态，多点集群，那么多点集群下一步的趋势将是完全分散状态。分散状态下的存储系统是什么呢？其实就是彻底的DAS状态，即存储架构又回到了每个使用者各自保有一个独立专供的存储系统，各自管理各自的存储系统，各个使用者互不干扰，却还可以取得良好的性能以及高容量，而且还不浪费多少电和物理空间。符合这种条件的存储介质是什么呢？当然是芯片存储而非机械存储。

当前的SSD硬盘，单块SSD的大块连续读吞吐量甚至可以超过350MB/s，写则超过了210MB/s；甚至在4K块随机读吞吐量也超过了200MB/s，写超过了180MB/s，随机读IOPS超过600，随机IO延迟不超过1ms。这确实是非常惊人的速度。这种速度，满足当前主流的应用系统已经不成问题。所以，随着技术的发展，大容量SSD的成本会逐渐降低到可以用得起的地步（128GB的SATA 6Gb/s的SSD价格目前为146GB的3倍还多），并且各种技术壁垒相继突破，一块或者几块SSD即可满足主流的应用，试问此时有何理由再去使用网络适配卡通过线缆连接到外部设备上去存储数据呢？

当存储介质有了一次质的飞跃之后，整个存储系统架构就产生了彻底的一次轮回。那么再之后会怎么发展呢？分久必合，当无线电技术发展一定程度时，数据再一次将会被集中存放，各个角色可以用无线通信来获取数据。那么再往后呢？合久必分，此时又怎么分呢？这个问题现在还无法想象。

集群存储系统可以分为基于Block协议访问的传统存储的集群，以及基于NAS协议访问的NAS集群系统，还有一类属于文件系统的集群，下面将一一介绍。

15.6.1 IBM XIV集群存储系统

IBM XIV存储系统为以色列的一家公司所开发，后被IBM收购。XIV是一种网络集群化存储系统，集群中的每个节点都是一台X86 Server，每个节点都包含12块本地SATA硬盘。

1. XIV系统的物理拓扑设计思想

共有两种类型的节点，一种是Interface Node，或者称Interface Module；另一种是Data Node，或者称Data Module。只有Interface Module上插有前端主机通道适配卡，比如FC卡、iSCSI卡等，这也是其名称的由来，意即主机端只能连接到Interface Module上。Interface Module本身也含有12块SATA磁盘。Data Module上没有前端主机通道适配器，只包含12块SATA硬盘以及两块双口1Gbps以太网适配器。每个节点有1颗Intel的4核CPU（新一代产品有2颗）和8GB的DDR2内存。

整个系统的连接拓扑图如图15-33所示。6个Interface Module通过两台冗余的以太网交换机与9个Data Module相连接，每个Interface Module

使用3条链路与一个以太网交换机连接，而Data Module使用两条链路与每个交换机连接。每个Interface Module包含4个4Gb/s的FC口，其中两个用于主机连接，另两个用于其他用途（Mirror、DR等）。

图15-33 XIV内部连接逻辑拓扑图

如图15-34所示为满配的XIV存储系统各种节点在机柜中所处的位置示意图。

图15-34 节点排列图

如图15-35所示为XIV存储系统实物的前视图和后视图。

图15-35 机柜实物图

如图15-36所示为XIV存储系统中节点机箱的透视图。

图15-36 节点机箱透视图

如图15-37所示为Interface节点的连接拓扑示意图。

图15-37 Interface节点连接示意图

如图15-38所示为节点的正视图。

图15-38 节点正视图

如图15-39所示为节点互连用后端交换机的实物图。

图15-39 节点互连交换机实物图

如图15-40所示为XIV存储系统的GUI配置界面。其个性化十足，仿

苹果操作系统菜单设计。

图15-40 GUI配置界面

2. XIV系统对LUN的分布设计思想

对于这样一个网格化的存储系统，卷/逻辑驱动器/LUN被设计为平均分布在多个或者全部节点之上，有一个映射图来记录每个LUN的分布状况。而且每个LUN对应的每个数据块都会被镜像一份，存储在与源数据块不同的任何一个节点上，源分块称为Primary，镜像之后的分块称为Secondary。这样一来，任何一个节点故障不会导致数据丢失，并且在任何一块磁盘或者整个节点故障之后，系统会根据LUN映射图来判断并将丢失的部分通过尚完好的源数据块再次镜像一份到其他任何一个节点上的剩余存储空间。

这种思想与NetApp公司的WAFL文件系统极为类似，大凡高度虚拟化的设备，其底层的逻辑卷/LUN都不是存在于固定位置的，XIV用映射图来遍历整个LUN在所有节点磁盘上的存储地址，并可以将LUN的某个块移动到其他节点并重新更新映射图，这种思想就是不折不扣的文件系统思想：文件系统可以将任何文件分步到硬盘上的各个空间，并使用Inode Tree来遍历整个文件的分布情况。WAFL和XIV正是对这种思想在卷管理设计上的灵活运用，但是XIV可能并没有沦为彻底的文件系统，其粒度以及元数据复杂度一定不如WAFL那么细。最终，一个LUN或者卷就体现为一个Filedisk，但是这只是对存储系统内部而言，对主机客户端而言一切都没有变化。如图15-41所示为XIV系统接受一个客户端主机发送的写IO请求之后的流程。

图15-41 写IO处理流程图

XIV系统在其某个Interface Module上接收到一个针对某LUN某地址段的写IO。

接收到IO的Interface Module将IO数据收入之后，首先通过Distribution Map判断这个LUN的这段地址落在的分块及其镜像块存在于哪个Module上。如果对应的分块或者镜像块其中一个是在自己本地硬盘中，那么这个Interface Module会立即将这份数据副本通过后端以太网发送到这个分块镜像所在的Module，然后通知客户端主机写入成功。这样，数据便会在两个节点的Cache中，实现了Cache Mirror，一旦这份数据尚写入硬盘的过程中，某个节点发生故障，那么还有另外一个节点保存着这份数据。

如果接收到写IO的Interface Module判断的结果是，这个IO对应的LUN分块没有存在自己本地硬盘，而是存放在了另外两个节点上，那么这个Interface Module会根据Distribution Map将这份IO数据发送到这个分块所对应的Primary副本所被保存的节点。

保存Primary分块节点收到IO数据之后，再次根据Distribution Map判断出对应这个分块的Secondary副本被保存的节点，立即将这个IO发送过去，当收到发送成功的回应之后，立即向刚才的那个Interface Module返回写入成功的回应。Interface Module接收到成功回应之后，立即向客户端主机返回写入成功的回应。

3. XIV系统的快照设计思想

Filedisk的设计思想会彻底地改变后续所有上层功能模块的设计，比如Thin Provision、Snapshot、Clone、Mirror、Dedupe等。例如，对于Snapshot的实现，WAFL和XIV都使用了Write Redirect的实现方式，这绝

对不是巧合，而是Filedisk的核心设计思想使然。

在WAFL中，每个Snapshot其实就是一份Inode Tree指针链条和其对应的实体数据块的留存影像，包括当前的活动文件系统，这种设计使得当前活动文件系统与Snapshot本身是同质化的，可以大大降低设计复杂度，从而提高运行效率。同样，在XIV的卷管理模块（本质是一个文件系统，而且是集群文件系统）中，每个LUN（文件）都对应一份Distribute Map（Inode Tree），如果需要将LUN（文件）中的某个或者某段数据块（文件的一字节或一段字节）移动到系统中的其他节点上的硬盘（文件所存储的底层硬盘空间），那么只需要将对应的数据作相应移动之后，在Distribute Map中作相应的指针改变即可。当快照生成时，采用Write Redirect方式，快照时间点的这个LUN的Distribute Map与其对应的实体数据将会被冻结，然后系统将当前的Map复制一份存放（或者只复制Map的跟入口，视设计不同而定），这个新Map就是当前活动Map。当随后有针对这个LUN的写IO请求进入时，系统将会把这个IO数据写到系统中任何一个节点的任何一个空闲数据块，并在新生成的Map对应的位置将指针更新，指向这个新数据块。如果再次生成快照，那么系统就把当前的活动Map再复制一份，然后将当前的Map冻结为快照Map，之后的动作以此类推。对于Write Redirect模式的快照具体设计思想可以参照本书第16章。

基于这个设计模式，XIV还可以复制快照，即生成一份与某个快照完全相同的快照，其原理很简单，就是复制一份这个快照的Distribute Map并冻结即可。

当然，对于一个粗线条的卷管理模块来讲，其管理LUN的数据粒度必定不会与彻底的文件系统相比，但是本质思想却是相同的。WAFL的管理粒度为4KB，而XIV的管理粒度为1MB。

4. XIV系统的故障恢复设计

经过上文的论述，我们可以看到XIV对数据分布的本质思想其实是分布式RAID 10。就相当于在一个文件系统之内将一个文件（LUN）复制了一份，并将副本存放于其他的位置（每个数据块的源和镜像不允许放在同一个节点上）。这样，整体系统的可用容量就相当于减半。我们可以将这种设计称为“上层分布式RAID 10”，那么与其对应的就是“底层固定式RAID 10”了。后者就是传统的RAID 10系统。传统的RAID 10系统有一个很大的不足，即不管Raid Group内分布了几个LUN，或者甚至有没有被实体数据所占用，那么底层均会将所有的数据块镜像起来。

刚才所说的“有没有被数据占用”，这句话或许有人不理解，RAID层怎么会判断其上的哪些块“有用”，哪些块“没用”呢？（关于这个话题请参考附录2的问题11和48）的确，正因为Raid无法判断，所以才需要把有用没用的一同镜像起来，这就是底层应该做的。负责判断有用或者没用的是上层文件系统，或者，就是我们当前论述的XIV的卷管理系统。底层实现不了这种判断的结果就是直接导致了资源浪费，在一块磁盘故障之后，系统做了太多的无用功，Rebuild了许多无用的数据块，实在是划不来。

上面讲了对应于“底层”二字，再来说说“固定式”的缺点。传统的RAID 10系统，源和镜像必须有相同的磁盘数量和容量，一对一，少一块都不行，而镜像端的磁盘容量可以比源端的大，但是多余的容量，RAID又会将其砍掉不用，很滑稽，这无疑是一个巨大的浪费。是否可以实现一种另外的模式呢？比如将一块磁盘上的数据，分开镜像到多块磁盘上，相当于把一块磁盘再RAID 0化；或者，将多块磁盘上的数据，镜像到一块磁盘上。对于后者，底层RAID是无论如何也实现不了的。

另外，底层固定式RAID 10要求Raid Group中的磁盘一定是在本地管理范围内，而不能跨计算机系统，如果本地计算机系统整体故障，那么整个Raid Group就无法访问了。

再来看看XIV的上层分布式RAID 10是如何解决上述这些问题的。

- 解决做无用功的问题。前文说过，XIV的卷管理系统是一个粗线条的文件系统，它当然可以感知自己所管理的文件（LUN）占用了哪块磁盘上的哪块空间，既然这样，那么卷管理系统就可以只镜像这些文件，而无须镜像硬盘上没有被文件以及元数据所占用的数据块。镜像操作是由卷管理层完成的，所以称之为“上层”。这就很好地解决了第一个问题。
- 解决磁盘数量必须一一对应问题。既然卷管理系统可以将LUN像对待文件一样将其复制到另外的存储空间，那么为何不可以将这个文件分开若干份存储于多个硬盘中，或者将原本分开存放于多个硬盘中的文件的各个部分再合并起来存放到一个硬盘中呢？当然是可以的了。比如，某个LUN被分为4块：B1、B2、B3、B4。这4块数据分别存放于D1、D2、D3和D4这4块硬盘当中。现在决定拿掉D1、D2这两块硬盘而保留这个LUN，那么卷管理系统会首先将B1、B2这两个块复制到D3和D4硬盘上，然后通知可以拿掉硬盘。此时，这个LUN就由原来分布于4块硬盘上变成现在分布于两块硬盘上了，当然这里只是一个RAID 0思想，并未涉及到RAID 10。

下面我们就来演示一下XIV卷管理系统是如何解决传统RAID 10模式下磁盘数量一一对应的问题的。

如图15-42所示，传统RAID 10模式下，设某个LUN被分为6块数据

块，分别标记为B1～B6，每个数据块的镜像块分别标记为M1～M6。这些数据分布在4块硬盘上，并且每块硬盘都有剩余空间。

图15-42 传统RAID 10

某时刻，D1磁盘故障，数据块B1、B2、B3丢失。但是其镜像M1～M3依然存在，此时，XIV系统为了恢复数据的冗余，需要将M1～M3数据块再次镜像到其他磁盘。如图15-43左边部分所示，XIV系统将这三个数据块镜像到了D3磁盘的剩余空间内。右半边显示的是另外一种镜像方式，即系统可以将数据块镜像到任何磁盘的任何位置。右半边所示的方式，同样可以保证数据的冗余性。

图15-43 D1故障

假如，XIV系统在D1磁盘故障之后，成功地将M1～M3再次镜像到了D3和D4两块磁盘之后，D3磁盘又发生了故障。那么此时可以判断出，数据依然没有丢失。为了恢复数据的冗余性，此时系统会再次将只剩下一份复制的数据块再次镜像到系统整体的剩余空间内。如图15-44所示。左右两半边分别对应了两种方式。

图15-44 冗余性恢复了（两种模式）

当镜像完成后，系统依然可以保证冗余性。如果此时D2或者D4任何一块磁盘故障，那么系统将没有冗余性，但是数据依然可以访问。当最后剩余的磁盘也故障时，数据就损毁了。某时刻，管理员手动添加了两块新硬盘，XIV系统会重新平衡数据分布，让LUN的6个分块平衡分布在这4块硬盘上。如图15-45所示为重新分布完成后的数据分布图。

图15-45 新盘加入之后数据重新平衡

从上面的例子中我们可以体会出，XIV的这种上层RAID 10设计，是一种可以随时迁移数据块的，并且可以任意分布式摆放数据块，不受磁盘数量限制而只受系统内整体剩余空间限制的新设计思想。所以我们在上文把这种模式叫做“上层分布式RAID 10”。

传统RAID 10系统在一块磁盘故障之后，只能将源磁盘上的全部数据镜像到新的磁盘中，如果此时系统已经没有新的磁盘，那么系统什么也不会做，只会报警通知管理员快增加新磁盘。

解决磁盘必须在本地的的问题。由于是上层RAID，所以XIV系统中的每个节点都可以通过后端的以太网交换机与其他节点进行内部通信，以同步所有LUN的Distribute Map，以及用于将本地存储的数据块镜像到另外节点上的磁盘。同一个节点中存储的数据块中任何一个数据块的镜像块都必须被存放在非本节点的任何一个或多个其他节点上。这样，一旦一个节点整体故障，数据依然可以访问，同时剩余的所有节点均会通过Distribute Map来判断故障节点中原来所存储的某个或某些数据块是否是本地存储的某个或某些数据块的源块或者镜像块；如果是，则证明这个数据块目前已经没有了冗余性，需要再次将其镜像，则系统会将存储于本地的这个或者这些数据块镜像一份，将其复制到另外的节点中的剩余空间存放。

目前所发布的XIV系统，最大配置为6个Interface Module+9个Data Module。最大磁盘数量=15×12=180块SATA硬盘。虽然系统整体容量不大，但是XIV的这种架构设计可以很容易地扩充到更多的节点。

此外，XIV的一项非常拿得出手的本领，就是，只要系统整体还有剩余的磁盘空间，那么只要磁盘一块一块的坏下去，甚至节点一个一个的故障下去，那么整体数据的冗余性依然会被保持，直到没有剩余空间

不足为止。当然，前提是，在一个磁盘或者一整个节点坏掉之后，必须等待系统Rebuild，也就是镜像完成之后，才能允许再坏下一块磁盘（与上一块磁盘在同一节点的磁盘可以一下全坏）或者下一个节点，否则数据很大几率上将被损毁。这一点，在经过上文演示之后相信读者都可以充分深刻地理解其本质了。

还有，XIV的另一项杀手锏就在于其在故障之后对系统冗余性的恢复速度方面。很多人都表示惊讶。据资料显示，“一个1TB容量的SATA磁盘损坏后，满配的XIV并且数据存满这块故障的磁盘后的Rebuild时间只需要30分钟左右，相对传统Raid恢复来讲，速度有了惊人的提升”。请注意，所谓“相对于传统Raid恢复”，是传统RAID 5还是传统RAID 10呢？这可能会让人产生思维定势，自然的考虑到RAID 5恢复时所耗费的时间，那是相当长的。但是XIV本质是RAID 10啊，得用传统RAID 10来比较才对，下文会做详细比较。

经过上面的分析之后，这种恢复速度丝毫不让人惊讶，因为其设计思想自然地就能够达到这么高的恢复速度，如果达不到，那才是让人惊讶之处。下面列出了几个让人不惊讶的原因及对应的分析。

原因1：看清楚，想清楚。是RAID 10不是RAID 5，如果换RAID 5试试？XIV的卷管理系统在故障恢复和数据分布时之所以可以设计得如此灵活，是因为其使用的RAID 10镜像的思想，而不是RAID 5的Stripe算XOR的思想。假如，XIV使用RAID 5的思想，也就是几个数据块做XOR，然后将校验值存放在其他存储空间。那这样的话，每当卷管理系统要将某LUN的某个块移动到其他位置的时候，系统都要在新位置上重算XOR，与传统RAID 5实现方式没有一点区别。所以如果使用RAID 5或者RAID 4的话，那么XIV就与常人无异了，就是一款普普通通的集群存储系统，其杀手锏也将不复存在。正因如此，由于RAID 5的XOR计

算需要耗费太多的计算资源和IO资源，所以XIV的恢复速度比RAID 5快十几倍也是很正常的。

原因2：分布式，理解这三个字。 传统RAID 10，一块磁盘丢失后，镜像的数据只能从那一块镜像磁盘上来读，然后还只能写到那一块新磁盘上去，也就是一个盘读，一个盘写。而XIV的卷管理系统是将LUN分块平衡存放于多个磁盘的，而这些分块对应的镜像块也都是存放在多个磁盘上的，所以在XIV进行Rebuild的时候，是从多个盘读，同时向多个盘写，相当于RAID 3的思想，众人拾柴火焰高，满配180块磁盘，所以恢复速度快是理所当然的了。

这里有必要再与RAID 5比较一下。假设同样有180块硬盘，组成了一个RAID 5，坏掉一块磁盘之后，系统需要从剩余的179块磁盘中读出其上的所有数据（当然数据是边读边运算的，这里将其转换为最终统计数字），然后再将运算结果写到新磁盘中。如果每块磁盘容量是1TB，内部磁盘环路带宽为4Gb=400MB/s，需要读出的总数据量为179TB，需要耗费的时间为 $(179 \times 1024 \times 1024 / 400) / 3600 = 130.3$ 小时，由于写入新盘和读取剩余磁盘为双工操作，所以不计入总时间，另外，也忽略了XOR运算所消耗的时间，而且磁盘环路带宽按照100%效率计算。这个数字与0.5相比，反差是巨大的。但是一定要清楚，RAID 10与RAID 5在恢复时间方面没有可比性。况且实际也不会使用这么多的磁盘来形成一个RAID 5组。实际中一般情况下就拿8盘RAID 5来算的话，Rebuild时间也要有8~9个小时，而且是系统外部负载很小的时候。

我们不妨再与传统RAID 10系统来比较一下。假设同样由180块硬盘组成一个传统RAID 10系统，其中90块盘与另90块盘互为镜像关系。某时刻，其中一块磁盘故障，此时系统会从这块磁盘对应的镜像磁盘将数据读出同时写入新磁盘。由于不牵扯任何XOR之类的额外运算，属于整

盘复制，IO类型属于连续读与连续写，所以可以按照SATA硬盘的理论连续读吞吐量60MB/s来计算。 $(1 \times 1024 \times 1024 / 60) / 3600 = 4.85$ 小时。可以近似认为180块硬盘配置的XIV在Rebuild时耗费的时间为传统RAID 10的1/10。如果今后XIV系统可以增加更多的节点，想必Rebuild速度会有所加快。当然，系统整体磁盘越少，Rebuild时间也就会相应延长了。

原因3：最重要的原因是，不做无用功！这也是3个原因中最重要也是效果最大的一个原因。恢复速度快，只重新镜像复制磁盘上已经被LUN所占用的那些数据块，而不是整盘镜像。如果这个坏盘上原先只有很少的数据块被占用，还用30分钟么？估计一分钟也可以了，甚至几秒钟都有可能，或者，根本不用时间，因为或许它上面本来就没有任何LUN去占用。

5. XIV系统很容易做到但尚未做到的

动态存储分级数据迁移是目前正在被热炒的几个存储系统上层技术之一。包括Compellent、EMC、HDS在内的多家厂商都已经在这个领域有相关的产品发布。但是IBM只是在Tivoli软件中实现了一个客户端基于文件的分层管理工具，并未在存储端提供基于卷或者基于Block的动态迁移方案。

XIV这种可以将LUN分块任意迁移到其他节点的功能，自然而然地就可以在稍加开发的基础上做到Tiered Storage Management。Compellent公司可以在其在线存储中实现基于Block的动态数据迁移，HDS可以在其存储系统中做到基于卷的手动迁移，EMC在其存储系统中提供FAST功能可以实现基于卷或者卷分块的动态迁移。可以看到，这些技术都是将整卷或者整卷细分成更细小的块来做迁移的。XIV原生的技术已经支持了卷分块迁移，相比于上述厂商，所需要开发的模块就少了很多。

遗憾的是，XIV目前尚未有相关技术发布。包括有类似潜质的WAFL，也未见动静。综上所述，XIV将LUN当作一个文件，然后在一个分布式多节点系统内利用分布式文件系统来管理这些LUN，再利用Interface Module将这些文件LUN以SCSI块的形式进行输出。

6. XIV系统的理论吞吐量计算

XIV存储系统在节点互连带宽方面不是很足。千兆以太网交换机，在15个节点满配的情况下，其后端整体互连带宽也不过 $6 \times 6 + 9 \times 4 = 72\text{Gb/s} = 9\text{GB/s}$ 。对于15个节点的存储集群，这个数值是比较低的，也是瓶颈所在。

由于系统在处理IO写的过程中，会产生2次或者3次IO传输过程，均会跨越后端交换机。我们在此给出一个理论计算推导出来的XIV满配情况下的最大读和写吞吐量的计算过程和结果。由于系统包含6个Interface Module和9个Data Module，所以需要读写的数据块地址落在前者类型节点之上的几率为 $6/15 = 0.4$ ，落在后者类型节点之上的几率为0.6。我们按照几率来进行计算。

1) 理论平均最大读吞吐量（Cache Miss）

假设，在一秒之内，系统接收到的所有读IO对应地址的Primary或者Secondary块均落在了Interface类型的节点之上。那么，我们假设IO类型为大块连续IO，每个Interface Module的12块SATA硬盘以额定速率发送数据，即每块硬盘60MB/s的速度，总速度为 $12 \times 60\text{MB/s} = 720\text{MB/s}$ ，6个节点后端额定速度共4320MB/s，前端额定速度为 $2 \times 6 \times 400\text{MB/s} = 4800\text{MB/s}$ ，所以整体理论额定速率以后端为准，为4320MB/s。

假设，在一秒之内，系统接收到的所有读IO对应地址的Primary或者Secondary块均落在了Data类型的节点之上，而且我们同样假设IO类型为大块连续IO。此时，每个Interface Module在接收到读IO请求之后都需要从后方的Data Module上取数据。所有6个Interface Module的后端链路总带宽为 $6 \times 6\text{Gb/s} = 36\text{Gb/s} = 4.5\text{GB/s}$ ，而所有9个Data Module所提供的链路总带宽为 $9 \times 4\text{Gb/s} = 36\text{Gb/s} = 4.5\text{GB/s}$ ，与前方Interface Module匹配，但是必须要看后端磁盘是否可以满足这个带宽。后端共 $9 \times 12 = 108$ 块SATA硬盘，每块理论最大带宽 60MB/s ，总后端磁盘带宽= $108 \times 60\text{MB/s} = 6480\text{MB/s} > 4.5\text{GB/s}$ ，所以，按照前端带宽为准，整体理论带宽为 4.5GB/s 。

按照几率比例将结果进行换算得出系统读平均理论最大带宽为：
 $(4.5\text{GB/s} \times 0.6) + (4320\text{MB/s} \times 0.4) = 4492.8\text{MB/s}$ 。除掉前端FC协议以及后端以太网与其上层协议传输耗费的带宽，按照15%来算，再除掉控制器处理所耗费的延迟资源等，按照10%计算，则最终实际结果应该接近于 3369MB/s 。

2) 理论平均最大写吞吐量 (Write Through)

- 假设IO类型为大块连续IO。由于后端牵扯到Cache Mirror过程，所以计算起来比较复杂。

3) IO对应的地址落在Interface节点 (几率0.4)

4) IO对应的块地址恰好落在接收到IO的Interface节点 (几率 $0.4 \times 1/6$)

5) 镜像块落在了其他Interface节点 (几率 $0.4 \times 1/6 \times 5/14$)

这种情况下，每个接收到IO的Interface节点可以用后端6条链路全速向镜像块所在的节点发送数据，即整体写吞吐量为 $6 \times 6\text{Gb/s} = 36\text{Gb/s} = 4.5\text{GB/s}$ 。

- 镜像块落在了某Data节点（几率 $0.4 \times 1/6 \times 9/14$ ）。

这种情况下，同样由于每个接收到IO的Interface节点只能向一个Data节点发送数据，则只能以Data节点后端链路数为准，即整体写吞吐量为 $6 \times 4\text{Gb/s} = 3\text{GB/s}$ 。

- IO对应的块地址落在了非接收到IO的Interface节点（几率 $0.4 \times 5/6$ ）。
- 镜像块落在了其他Interface节点（几率 $0.4 \times 5/6 \times 5/14$ ）。

这种情况下，要计算理论最大带宽，需要让所有的节点在收发数据时不发生冲突，即每3个Interface节点为一组（接收到IO的节点、一次转发节点、二次转发节点），共两组。整体传输带宽为： $2 \times 6\text{Gb/s} = 1.5\text{GB/s}$ 。

- 镜像块落在了某Data节点（几率 $0.4 \times 5/6 \times 9/14$ ）。

这种情况下，要计算理论最大带宽，需要让所有的节点在收发数据时不发生冲突，即每两个Interface节点为一组（接收IO的节点、一次转发到的节点），共3组。但是这三组接收传送组的终点是各自对应的Data节点，所以按照Data节点的最大链路来计算。整体吞吐量为 $3 \times 4\text{Gb/s} = 1.5\text{GB/s}$ 。

将以上所有结果乘以各自对应的几率，我们得出第一种情况的吞吐量贡献值为 0.736GB/s 。

- IO对应的地址落在Data节点（几率0.6）。
- 镜像块落在了Interface节点（几率 $0.6 \times 6/14$ ）。

在这种情况下，一个接收转发组由接收到IO的节点、一次转发Data节点和二次转发Interface节点组成。二次转发Interface节点又可作为接收到IO的节点，依次类推串联，每个转发组都不会与其他转发组冲突。这样一共是5.5组。以Data节点链路数量为准计算，整体吞吐量为 $5.5 \times 4\text{GB/s} = 2.75\text{GB/s}$ 。

- 镜像块落在了其他Data节点（几率 $0.6 \times 8/14$ ）。

在这种情况下，一个接收转发组由接收到IO的节点、一次转发Data节点和二次转发Data节点组成，共可组成4.5组。以Data节点链路数量为准计算，整体吞吐量为 $4.5 \times 4\text{GB/s} = 2.25\text{GB/s}$ 。

将以上所有结果乘以各自对应的几率，我们得出第二种情况的吞吐量贡献值为 1.478GB/s 。

将第一种和第二种情况的贡献值相加得出系统整体理论写吞吐量为 2.214GB/s 。除掉协议层耗费的带宽以及控制器处理延迟总开销共25%，得出最后实际写吞吐量应接近 1.66GB/s 。

XIV系统在读和写同时进行的时候，由于后端链路会发生严重的冲突（读和写都需要后端链路进行发送和接收动作），所以理论值更是达不到上述分析结果了。其次，由于多个LUN都均匀分布于所有节点的所有硬盘，所以，在多LUN并发连续大块IO读或者写的时候，很有可能会造成硬盘寻道冲突，大大降低吞吐量（这个冲突作用我们会在性能优化章节中介绍和分析）。而对于小块随机IO，即OLTP类型的IO，分布式的LUN的设计会随着节点的增加而IOPS正比升高。

如图15-46所示为XIV系统的吞吐量实测值，我们看到与推导值较为接近，但是误差比较大，这个无法避免。

图15-46 吞吐量实测值

如图15-47所示为XIV系统的IOPS实测值（Cache Miss），可以看到这个值对于一个由180块SATA盘组成的集群系统来讲还是比较可观的。

图15-47 IOPS实测值

7. 全打散式LUN分布方式的致命缺点

当遇到多主机用大块连续IO的方式并发访问多个LUN的时候，此时系统整体的性能会骤降，具体原因可以参考本书后面章节。机械磁盘最怕寻道，加上XIV使用的都是SATA盘，平均寻道速度更慢，此时便是XIV最难受的时候。

15.6.2 3PAR公司Inserv-T800集群存储系统

1. Inserv-T800系统硬件架构简述

3PAR 公司设计的Inserv-T800为一款X86集群式存储系统。整个集群系统可以由2~8个节点组成，其中节点必须成对添加，每1对为1个单元，这1对节点共享的带起后端的两串磁盘扩展柜（每串4个），一个节点对不能直接访问另一个节点对所管理的磁盘。

每一个节点都是一台中高配置的X86 Server。系统内的所有节点间两两直通，每两个节点都使用一条独立的100MHz 64bPCI-X总线相连。在一个8节点的系统内，1个节点需要与另外7个节点互连，所以，每个

节点都需要7条PCI-X总线连接到背板，形成一个具有28条直通PCI-X总线的星形结构，如图15.48右侧所示。但是由于PCI-X为半双工传输，所以整体系统内部互联带宽为 $28 \times 800 \text{MB/s} = 21.875 \text{GB/s}$ 。由于PCI-X并非一种包交换网络传输模式，所以背板并不需要加电，是一块布满导线和接口的被动式的背板，PCI-X编解码和传输电路都位于节点内部。同样是由于被动背板的原因，如果系统节点数量达不到8个，比如只有两个，那么此时只能用到后端的一条PCI-X线路作为互联，因为背板没有智能到判断连接的节点数量并自动将电路开关切换以便让当前连接的节点尽可能多地使用后端的所有总线。

如图15-48所示为单个节点内部的架构。每个节点使用两个双核Intel的CPU，并使用4GB内存作为3PAR的InForm操作系统的运行空间。另外加一块ASIC芯片来负责数据在前端和后端之间的传输、与其他节点的缓存镜像操作、RAID XOR运算、节点间相互通信以及ThinProvision和LUN Shrink功能。这款ASIC的功能是非常多的，而且最新奇的是，ThinProvision以及LUN Shrink的功能也被内嵌到了这款芯片中来执行，这在其他厂商是从未见过的，也是3PAR的一大亮点。目前这款芯片最新的一代为第三代。芯片内部有3个133MHz 64b的PCI-X总线用于连接前端和后端的接口卡；以及7个100MHz 64b的PCI-X总线专门用于连接其他节点。ASIC芯片直接控制着12GB的数据缓存。

图15-48 控制器架构示意图

如图15-49所示为节点控制器的实物图。可以看到共有6个PCI-X扩展卡插槽，它们共享3条PCI-X总线。图15-50左侧所示为系统的背板，可以看到其上的8个高密度针接口对应着8个节点。图15-50右侧所示为磁盘扩展柜实物图。3PAR的扩展柜很有特色，在4U的空间内放置了40

块3.5英寸硬盘，其原因是由于磁盘竖置并且紧密排列并且解决了散热和共振问题，这在其他厂商产品中也是没有过的。并且，节点后端的每个FC Loop仅连接一个扩展柜，即40块硬盘。扩展柜同样也是双Loop结构，一个节点对中的每个节点各拿出一个后端FC接口来连接一串（4个级联）扩展柜，每个节点再拿出另外一个FC口连接对方节点的扩展柜。这样，每个节点对最多可以共同连接8个扩展柜，即320块盘，4个节点对（8个节点）共可以连接的最大磁盘数为1280块。

图15-49 控制器实物图

图15-50 背板和扩展柜实物图

图15-51所示为8个节点与各自扩展柜的连接情况示意图。每个节点对最大可以连接8个扩展柜，图中未标识出。

图15-51 8节点互联

图15-52所示为Inserv-T400系统，最大4节点系统的机柜，机柜上部为磁盘扩展柜，下部可以放置4个节点控制器。左侧的背板为8接口，是对应T800的，这里只是示意图。T400使用的是4接口的背板。

图15-52 T400系统示意图

2. Inserv集群存储系统架构简评

Inserv集群存储系统是一款特点很多的产品。

首先，它使用直连的廉价PCI-X总线作为节点间互连的通道，但是

由于PCI-X半双工的限制，使得8节点间互连带宽只有21.875GB/s。而且由于采用点对点直连而非交换方式，这就造成系统的扩展性大受限制，N节点的系统就要求每个节点提供N-1条PCI-X总线，不适合扩充到更多的节点。

其次，节点内部使用专用ASIC来实现大部分数据操作，并还负责执行ThinProvision和LUNShrink（或者叫LUN Space Reclaiming）的功能，这又会大大降低系统主CPU的负担。再次，磁盘扩展柜的高密度设计，大大减少了空间占用以及连线，一定程度上也降低了耗电。如图15-53所示为这款ASIC芯片的实物图。

图15-53 Thin ASIC

Inserv集群存储系统的一个比较显著的瓶颈点在于每个节点使用3条1GB/s的PCI-X总线来支撑6个扩展卡共24个4Gb/s的FC接口，这实在有点捉襟见肘。这也注定了这款产品在满配8个节点，前端128个4Gb/s的FC接口的情况下，系统整体带宽吞吐量上不去，停留在不到6.5GB/s的级别上。

但是系统整体的IOPS吞吐量是很不错的，如图15-54所示，SPC-1测试取得了224990 IOPS的成绩。而且曲线非常平滑和趋缓，显示了系统整体极高的IO消化能力。

图15-54 IOPS SPC-1值

15.6.3 EMC公司Symmetrix V-MAX集群存储系统

1. Symmetrix V-MAX集群存储系统硬件架构简述

Symmetrix V-MAX是EMC在2009年中旬推出的一款集群存储系

统。其使用X86 Server作为节点，每个节点配备两颗四核Intel CPU以及几十GB的内存，若干扩展卡，另外当然也少不了用于节点互联的网络通道卡——RapidIO适配卡。V-MAX集群采用RapidIO网络来作为集群内部互联通道，整体采用两个外部独立RapidIO交换机来提供交换介质通道。

如图15-55所示为V-MAX系统满配时的拓扑图，共16个Director（节点），每两个Director组成一对共同挂起后端一串磁盘扩展柜。每对Director被称做一个Engine，满配最大8个Engine。如图15-55右侧所示为单个节点（Director）内部的带宽情况。节点硬件指标方面，这里就不作过多描述了，因为x86 Server都差不多。这里主要描述一下RapidIO这个网络。

图15-55 满配时示意图

图15-56所示为单个Engine的物理视图。两个Director组成一个Engine并放置于同一个机箱之内。

图15-56 单个Engine物理视图

RapidIO是一种高速网络，目前被广泛用于嵌入式半导体器件之间的互连。像任何高速传输网络协议集一样，RapidIO也定义了一套自己的传输协议层次，在OSI各层都有相关定义。物理层可以采用并行传输或者串行传输，但是目前广泛被应用的是串行传输，8/10b编码机制，目前单向传输速率为3.125GBaud/s，合2.5Gb/s=312.5MB/s。同样可以4条串行链路捆绑形成4X链路，加上全双工设计，一条4X的RapidIO链路可以提供双向20Gb/s=2.5GB/s的带宽。链路层采用成帧传输，具有链路层流控和传输层流控机制。网络层，RapidIO定义了8b或者16b的地址用

于路由/交换每个Packet。传输层，RapidIO根据不同应用行为规定了多种传输类型。相比Fibre Channel来讲，RapidIO协议更加高效。目前最多支持一个交换域内16个端点，所以这个限制也是目前V-MAX存储系统的节点限制，即16个节点。

RapidIO目前已经成为一种标准开放的芯片通信网络标准，有多家厂商参与了这个标准的制定和维护，比如德州仪器、朗讯、EMC等。RapidIO下个速率级别为6.25Baud/s，合单向20Gb/s的速率。

RapidIO提供对所有节点共享内存的支持，即在一个RapidIO网络之内的所有节点上的内存共同组成一个逻辑的大内存空间，每个节点都可以寻址这个全局内存。当目标地址位于本地内存的时候，数据请求无须经过外部网络；当目标地址位于其他节点的内存的时候，那么本地节点的数据请求会经过外部RapidIO网络被发送到对应的节点并将对应数据取回。如果本地寻址不命中，则由于经过了外部网络，所以整体响应时间就会增加。

Symmetrix V-MAX系统就是一个共享全局内存的集群存储系统。

2. Symmetrix V-MAX集群存储系统IO处理流程示例

每个Director的内存被分为三大区域：CS（Control Store）、GM（Global Memory）和S and F（Store and Forward）。CS区域保存着运行在每个Director上的操作系统所必需的运行时数据；GM区域用于存放被缓存的写IO数据或者被Prefetch的待读IO数据；S and F区域用来存放Director后端之间相互等待交换的数据队列。下面我们通过几个示例来了解这些逻辑区域的角色。

1) 1个**Director**参与，本地**Read Hit**

如图15-57所示，某Director某时刻接收到了一个主机发送的读IO请求，Director在接收到这个IO请求之后，查询系统全局内存中是否有针对这个IO地址的缓存数据存在。本例中，Director恰好发现对应的缓存Slot就本地的GM中，所以Director在S and F区域生成一条针对这份数据在GM区域中的指针，然后追加到S and F区域队列尾部等待发送。随着S and F区域队列不断被执行，这份数据被发送到主机客户端。

图15-57 IO流程1

2) 两个**Director**参与，**Read Hit**于远程**Director**

如图15-58所示，Director1接收到主机的一个读IO请求，首先查询系统全局GM是否有对应的缓存Slot，发现在远程Director2的GM中存在针对这个IO地址的缓存数据Slot，Director1立即通过后端RapidIO网络向Director2发起请求，将Director2的GM中对应的数据通过RapidIO网络传送到Director1的S and F区域队列中。数据收到后，随着Director1的S and F区域队列不断被执行，这份数据最终被发送到主机客户端。

图15-58 IO流程2

3) 3个**Director**参与，**Read Miss**

如图15-59所示，Director1接收到主机的一个读IO请求，首先查询系统全局GM中是否有针对这个IO地址的缓存Slot，发现系统全局GM内没有针对这个地址的缓存Slot，所以Director1会为这个IO对应的地址分配一段缓存，此时Director1会根据某种策略，在集群内任何一个节点的GM上都可以进行分配。本例中假设Director1选择了在Director2的GM中

为这个IO对应的地址分配缓存，则Director1会通过RapidIO网络向Director2发起这个命令请求。Director2收到命令后，在本地GM为这个IO地址分配对应的缓存，并同时根据IO地址来判断这个IO所请求的数据存在于哪个节点的磁盘上。本例中假设数据存在于Director3的磁盘上，则Director2向Director3请求这份数据，Director3收到请求后，从其后挂的磁盘中将这份数据读出并存入本地S and F区域等待被发送。Director2收到Director3发送的数据后，将其放入已经分配的位于GM中的缓存Slot中，并在本地S and F区域队列尾部追加这份数据在GM中的指针。最终，队列中这份数据被发送到了Director1，然后被发送到主机客户端。

图15-59 IO流程3

上面这个例子中所述的过程可能有点让人琢磨不透，即为何此时Director2会参与进来，在发生Read Cache Miss时，系统为何不直接根据IO对应的卷和卷中的地址来判断所请求的数据落在哪个节点管理的后端磁盘中。如果直接在包含IO请求数据对应磁盘的那个节点来分配缓存，那么就会节省一次多余的数据传送过程，节约后端网络带宽。对于这个问题，后文会有阐述。

4) 4个Director参与，Write IO

如图15-60所示，Director1接收到主机的一个写IO操作，Director1根据这个IO针对的卷和卷中的地址判断，对应数据的磁盘不在本地，所以立即在系统全局GM中分配两份缓存以用于保存这份IO数据。本例中，Director1选择了在Director2和3的GM中各分配一块缓存用于存放这份数据，Director1将数据指针追加至本地S and F区域队列准备发送到Director2和3为其分配的缓存中。至此，这份写IO数据成功的在系统内部保留了两份，避免了单点故障，可防止一旦数据未写入硬盘之前整个

节点发生故障所造成的数据丢失。Director在完成缓存镜像的操作之后，立即向主机返回写入成功的消息。之后，Director2和3中的一个（Primary镜像），比如Director2，根据IO针对的卷和卷中的地址判断出此IO的目标数据对应的磁盘存在于Director4上，所以Director2将数据指针追加到本地S and F中，将数据发送到Director4的S and F中，Director4收到数据之后将其写入后端对应的硬盘中。

图15-60 IO流程4

对于写IO的处理，V-MAX也同样可能产生额外的不必要操作。比如上个例子中，数据完全可以只在接收到IO的Director即Director1与数据所保存的磁盘所在的Director即Director4的GM中保存两份即可，当数据写入硬盘之后，对应GM中的缓存Slot自然变为Prefetch类型的缓存。但是上个例子中，数据被多余的传送了两次。

当然上面所有示例中的情况都是最极端的情况，实际中有可能接收到IO的Director缓存将被耗尽，或者处理资源耗尽，或者由于各种原因，导致缓存Slot必须在其他节点分配，这种情况是有可能发生的，一旦发生，就需要额外的操作来解决这些问题；同时，额外的操作使得系统IO响应时间延长，相应主机端IO的压力就会随之降低，使得系统资源状况得以恢复，然后再次轮回发生，这也在一定程度上做到了压力自缓解。

对于一个集群存储系统，主机理应可以连接到集群中任何一个节点来向整个集群中存在的卷发起IO请求而不关心这个卷对应的实际数据存放在哪个节点上或自己是否就正在向这个节点发送IO请求。当主机将IO发送到一个其所挂的后端磁盘中并不包含对应数据的节点的时候，就要求节点通过后端互连通道进行数据转发动作了，这一点是无论如何不可

避免的。因为不可能强制让主机连接到磁盘上有对应数据的节点，这样的话就失去了集群的意义。况且，有些集群比如XIV会将卷数据分块存放于集群中的每个节点的磁盘上。虽然集群内部数据互转在所难免，但是我们依然需要考虑节约后端有限的互联带宽。

V-MAX的这种设计使得缓存Slot可以脱离数据对应的磁盘所依附的节点而存在于任何节点上，游离于各个节点之间，耦合非常松，有很大的灵活性和压力自缓解功能。当系统资源压力不大的时候，系统应当选择最优的缓存分配方式以降低IO延迟和节约后端互连通道带宽。

V-MAX的这种设计可以说与操作系统内核使用Page Cache机制（见本书后面章节）是同源的，操作系统可以将各种实体比如文件、设备映射到内存空间中，访问内存空间就等于访问文件或者设备对应的地址。实际上，V-MAX采用某种hash算法（比如DHT），根据IO的发出主机、目标地址、长度等信息，做一致性hash计算，从而映射到集群中的某个节点，对应该目标地址的缓存一定位于某固定节点中固定的内存区域，使用这种算法可以避免每次IO都查表来确定其缓存位置，能够节省计算资源，提升速度，但是却可能由于多次转发而降低速度，这是个矛盾。其思想与缓存与RAM之间的映射类似，都各有收益和牺牲。

FC SAN集群存储系统小论： EMC与3PAR之间一直火药味比较浓，EMC长期以来一直诟病3PAR的PCI-X总线、不支持8Gb的FC和10Gb的以太网、不支持SSD和不支持分级迁移等。

3PAR也时常调侃EMC的一些策略。HDS和EMC之间也在互相拿对方的架构指指点点。其实本来就没没什么可指点的，表现上可能不一样，但是骨子里却都一个样。

这次，EMC来了个大翻转，一改DMX的直连矩阵架构，全面转向基于包交换互联网络的集群存储架构，一改PowerPC，全

面转向开放的Intelx86。仔细的读者可能会发现，V-MAX甚至与3PAR的Inserv-T系列集群架构有些许相似之处，即节点都是成对出现。3PAR和V-MAX之所以必须以节点对的形式出现，是为了使用这一对节点共同连接一串磁盘扩展柜，以避免单点故障。比如，一旦某个节点Down机故障，如果其后挂的扩展柜没有另外节点来接管的话，那么所有存储于这串扩展柜上的数据将无法再被访问到。这里不得不提一下XIV，XIV的镜像保护方式和动态随时迁移数据的设计，让其可以只用一个节点连接后端的磁盘，虽然它目前还没有扩展柜，但是不见得未来不支持扩展，毕竟X86 Server是个很开放的架构，只要插上一堆扩展卡，没有做不到的事情。

V-MAX在节点内部使用了PCI-E，8Gb的FC和10Gb的以太网，节点互连通道使用了基于包交换的RapidIO网络，这些用料比起3PAR来可是足足高了不少。

目前HDS公司尚未发布集群存储系统，并一贯坚持着其基于Crossbar的大型主机架构控制器。随着IBM、EMC这两家巨头相继发布了纯集群存储系统，不知道这三家巨头中的最后一家HDS还能坚持多久。

目前X86系统越来越普及，性价比越来越高，作为开放系统，逐渐地被存储厂商用于新的集群存储系统上来。而存储厂商可以基于这些开放系统，研发更加高级、更加智能化和移植性良好的存储软件模块，可能这就是激发X86集群普遍被应用的原因之一。

存储系统中，硬件的发展前景已经变得缓慢，而真正更加需要的，不是数据存储方面，而是数据管理方面。

集群NAS系统和集群文件系统：前面介绍并分析了几个基于Block SAN方式访问的集群存储系统。另一大阵营，即NAS存

储系统，一样可以集群化，而且已经变成了一种趋势。促成集群NAS发展的一个最大原因就是因为在NAS系统前端以太网速率较低，传统NAS设备单个控制器性能不高，能够满足的吞吐量也很低，导致传统NAS存储系统常用于二线应用，比如文件共享或者一些非关键而且对性能要求不高的应用。然而，一些特殊的应用，需要多点共同读写某个文件的应用系统，比如3D渲染集群、电影渲染集群等，这些集群中包含几百甚至上千台服务器节点，它们可能需要同时读或者写某个大文件，这种需求如果使用Block级别的存储系统的话，那么务必需要在应用程序中引入文件锁机制，就会大大造成应用程序设计的负担并且不具有通用性，每种应用都自行设计集群模块，是没有必要的。此时，需要一个公共的集群层，而NAS（NFS、CIFS）自身就已经有文件锁机制，所以只要应用程序在设计的时候调用对应NAS协议提供的锁API即可。

苦于NAS系统的性能问题，在客户端节点过多而NAS服务端性能又太低时，系统整体的性能将会很差。这就迫切需要NAS存储系统进行集群化扩展，将客户端的IO压力分摊到集群中的每个节点上。除了将NAS系统集群化之外，还有另一种选择，即集群文件系统。

下面介绍几种集群NAS系统或者称集群文件系统的设计架构。

所谓“谁才是真正的**Scale-Out**？”：IBM自从亮相了XIV之后，EMC接着出了V-Max，接着HDS也推出了VSP。这三者都宣称自己是Scale-Out架构，在业界也引发了一些讨论，有人认为只有XIV才是真正的Scale-Out，而V-Max与VSP则不算Scale-Out。对于这个问题，我是这么看的。

大家知道服务器多CPU架构变迁过程，一开始是单CPU，后来发展到双CPU或者多CPU的SMP架构，也就是多CPU共享相同

的内存、总线、操作系统等资源，每个CPU访问全局内存任何地址耗费的时间都是相等的。还有一类AMP架构，即不同CPU做的事情是不同的。但是由于共享访问冲突，SMP架构扩展性-效率曲线已经达到瓶颈。为了进一步提高CPU数量的同时保证效率，NUMA架构出现了，也就是将多个SMP进行松一点的耦合，多个SMP之间通过CrossBar Switch高速交换矩阵互联，每个SMP都有各自自己的内存，一个SMP内部的CPU访问自己的内存时与之前没什么两样，但是要访问其他SMP处的内存，就需要走交换矩阵，导致延迟增加，所以，NUMA通过牺牲了内存访问的时延来达到更高的扩展性，比如可以将数百个CPU组成NUMA架构。SMP和NUMA架构对于软件程序方面的影响不大，同一台主机内都使用单一操作系统。但是由于NUMA访问远端内存时的时延问题，导致NUMA架构下的效率也不能随着CPU数量的增加而线性增长，只是比SMP要好罢了。此时，MPP架构就出现了。MPP可以说已经与CPU已经关系不大了，MPP说白了就是将多台独立的主机使用外部网络来组成一个集群，显然MPP架构下，每个节点都有各自的CPU、内存、IO总线和操作系统，属于最松的耦合，而且运行在MPP集群中的软件程序的架构也需要相应改变，变为大范围并行化，并尽量避免节点之间的消息传递。由于软件程序发生了变化，那么MPP的效率随节点数量的增长就可以呈线性关系了。其实，如果在NUMA架构下，软件也可以避免尽量少读取远端内存的话，那么NUMA效率也会线性增长，但是NUMA架构下的操作系统仍然是同一个，内存仍然是全局均匀的，而程序架构又尽量保持不变，那么就不可避免的时不时访问远端内存了。MPP相当于把内存强制分开，把操作系统强制分开，把程序架构也强制改变从而保持海量计算下的效率线性增长。

那么再说回到存储系统。与服务器CPU架构演进相同，可以把存储系统的控制器类比为CPU，而后端磁盘柜类比为一条条的内存。一开始的单控，后来的双控互备份（传统双控存储），一直到双控并行处理（目前只有HDS的AMS2000存储系统为双控并行架构），到这个阶段就类似于AMP（双控互备）和SMP（双控并行）架构，后来则有多控并行对称处理架构，Oracle的RAC集群也可以视作一种多点SMP，各种共享底层存储的集群文件系统及基于这种文件系统所构建的存储系统也属于多点对称SMP。

同样，由SMP到NUMA的过度也出现在了存储系统中，比如EMC的V-Max，相当于多个SMP（一对控制器组成一个Director等价于一个SMP矩阵）利用高速交换矩阵（RapidIO）来共享访问每个SMP上掌管的内存。

由NUMA到MPP的过度一样也出现在存储系统中，IBM的XIV就属于松耦合MPP架构，多个节点之间彻底松耦合，各自都有各自的CPU/内存/总线/磁盘/IO接口，使用外部以太网交换机，使用TCP/IP协议互相通信。而HDS的VSP则更像一个紧耦合的MPP，MPP对软件架构变化很大，所以传统存储厂商很难将之前的架构演变到MPP上来。另外一种属于MPP架构的存储系统就是各种分布式文件系统（注意，并非共享存储的集群文件系统）。

至于谁才是真正的Scale-Out，这个是个无定论的问题了。

SMP/NUMA/MPP其实都算Scale-Out，只不过程度和形态都不同罢了。有人说MPP才是真正的Scale-Out，可能是基于MPP流行的原因。但是不能一概而论。MPP架构的存储，例如XIV，由于特定场景下，由于单路IO就可能导致整个MPP集群中的磁盘资源全部牵动（每磁盘同一时刻只能执行一个IO），在多路

大块连续IO并发的情况下，反而效率很差（比如多流大块连续地址IO）；而某些特定场景下，多路IO之间牵制很少，则表现出线性增长的性能（比如小块高随机IO）。这也可以类比为将一个程序并行分解成多个执行颗粒（类比为高随机IO），颗粒间的关联性越少，则节点间通信量就越少，则并行执行的效率越高，一个道理，所以MPP自身为Share-Nothing架构，那么运行在它上面的程序颗粒之间最好也Share-Nothing。对XIV的具体分析可以参见后面的一节。

SMP、NUMA和MPP各有各的好处，也各有各的应用场景。比如SMP适用于扩展性要求不太高而又不想对程序改变太大的场景，而MPP则使用海量数据下的高扩展性需求场景，需要对程序有较大改变才能获得良好性能。同样对于存储也是这样，比如一旦决定用MPP架构的存储，那么就需要面对多流大块连续IO场景下性能不佳以及效率-扩展曲线的线性不佳这两个事实。或者你去修改上层应用，将大块连续IO改为高随机IO，而这显然荒唐。并且为了适应存储去修改应用，这一般是不可能被接受的。而MPP架构却被广泛用于互联网运营商的底层Key-Value分布式数据库，其高随机小块读访问场景下能获得巨量的性能以及线性的效率-扩展曲线。

15.7 集群NAS系统和集群文件系统

15.7.1 HP公司的Ibrix集群NAS系统

IBRIX是一家专做集群文件系统的公司，近年来被HP收购。IBRIX的集群文件系统称为“Fusion”，其架构中包含Fusion Segment Server/Client和Fusion Manager。下面了解一下IBRIX的架构。

如图15-61所示为IBRIX Fusion 集群文件系统部署之后的整体系统架构示意图。Fusion Segment Server是IBRIX集群的主体软件包，需要将其安装到集群中的每一台PC Server中。集群中的所有节点从后端的磁盘阵列中获取存储空间以便存放文件，这些存储空间可以被所有节点共享，也可以单个节点独享。集群中的节点也可以使用本地存储空间或者DAS直连存储的存储空间来存放文件。推荐使用共享存储空间，因为一旦某个节点发生故障，那么系统会自动将共享的存储空间挂载到正常节点之上继续提供服务。客户端主机可以使用三种方式来访问Fusion集群：NFS、CIFS、Fusion Client。集群中的每个节点都有各自的不同的IP地址，客户端只要访问任何一个节点即可访问到集群内所有的数据资源。集群内的节点各自有各自所管理的文件系统和Mount Point，但是每个节点都可以向外部Export所有节点上的Mount Point（Export其他节点的Mount Point之前，本地节点中必须提前也创建这个Mount Point。当然也可以手动控制那些只有Export自己管理的Mount Point），客户端如果一旦试图访问不受本地管理的数据，那么这个节点就会从管理这份数据的那个节点将这份数据对应的Metadata拿过来，然后通过Metadata来获取这份数据在后端存储空间内的位置，比如哪个Volume，哪个磁盘，然后通过后端存储网络将数据读出来再返回给客户端。如果使用非共享存储，则收到IO请求的节点会通过前端以太网从保存对应数据的节点把

数据拿到本地然后返回给客户端。

图15-61 IBRIX整体拓扑图

客户端可以采用NFS或者CIFS方式访问集群节点，也可以安装一个Fusion Client代理程序从而通过这个代理来访问集群节点。NFS和CIFS方式下，如果某节点所连接的客户端试图访问一个不受本地管理的目录或者文件的话，就会发生上文所述的过程，需要耗费一定的开销。但是如果使用Fusion Client代理来访问集群的话，Fusion Client会预先从集群中将Metadata拿到客户端缓存，这个Metadata描述了哪些Mount Point受哪些节点管理，所以，Fusion Client会将客户端发出的访问请求转发到集群中实际管理被请求数据的那个节点。这样，虽然客户端只显式挂载了集群中一个节点上的Mount Point，但是Fusion Client却会在底层隐式地将所有请求对号入座转发到对应的节点，这样就避免了节点之间互相要求Metadata的开销，系统性能有所提升。

如图15-62所示为上图中所示的一个Fusion Segment Server内部的软件层次架构图。Segment Server首先包含了一个Cluster Aware Logical Volume Manager，即CLVM。由于集群中多个节点可以共享后端存储，所以这里的卷管理软件也要支持多节点共享卷。其他功能与普通LVM无异，LVM先将物理LUN组成VG，然后在VG中最终做成的Logical Volume即LV，被Segment Server称为一个“Segment”，即分段的意思。每个Segment必须只隶属于一个Segment Server管理，但是一个Segment Server可以管理多个LV，即Segment。当某个节点故障之后，正常的节点可以直接接管故障节点原先所管理的Segment，并且接管故障节点原本承担的所有前端数据访问（如果某个Segment存储空间为非共享的，那么管理这个Segment的节点故障后，正常节点无法接管）。Segment之上便是IBRIX Fusion文件系统，Fusion FS对Segment进行格式化，并且

将Segment挂载到一个Single Name Space路径之下，每个节点的挂载动作都会通知到集群中所有节点以避免冲突和统一口径。集群内的通信是通过以太网进行的，后端数据路径则通过后端SAN交换机或者本地存储和DAS直连存储，后端的SAN可以是FC方式也可以是iSCSI方式，本地存储或者DAS的话可以通过RAID卡挂JBOD或者本地磁盘等。

图15-62 Segment Server内部架构图

在Segment上创建文件系统的时候，可以只使用一个Segment作为这个文件系统对应的存储空间。然而，也可以使用多个Segment作为这个文件系统的存储空间，每个Segment又可以隶属于单一或不同的Segment Server来管理，文件和目录在多个Segment之间的存放策略由Allocation Policy模块管理，其提供了多种Polic，比如ROUNDROBIN（新文件或者目录轮流地在各个Segment之间存放）、STICKY（新文件或者目录都存放到某个固定Segment中直到这个Segment的剩余容量达到一定阈值）、DIRECTORY（新文件或者目录将被存放在与其父目录所在Segment相同的Segment上）、LOCAL（将新文件或目录存放到接受客户端请求的节点所管理的Segment中）、RANDOM（新文件或者目录随机选择一个Segment进行存放）等。

这些策略可以让管理员充分调节以平衡系统的负载。如果客户端使用Fusion Client来访问集群，由于其可以直接与所有Segment Server通信，所以这些策略必须放到Client端执行；如果客户端使用NFS/CIFS协议来访问集群，则由于客户端只能直接访问它所Mount的那个Segment Server，所以Policy必须在所有Segment Server上执行。Segment有三种类型，第一种是只可以存放目录的，第二种是文件目录都可以存放的（Mixed），第三种是只可以存放文件的，除非有特殊用途，默认情况下皆使用Mixed模式。

Segment的Owner可以动态地从一个Segment Server迁移到另一个，迁移过程对前端访问没有影响。由于集群底层使用了CLVM，在一个节点上创建的Segment，在其他节点上都会同步显示。在选择一个或者几个Segment建立文件系统的时候，可以指定用何种Policy以及将哪个Segment分配给哪个节点。Segment的大小可以不同。如果使用多个Segment来作为某文件系统的存储空间，那么在创建文件系统的时候可以指定其中哪个为Root Segment，谁管理Root Segment，谁就是掌控这个文件系统Mount/Unmount的首席执行者。

文件系统被创建之后，需要在Segment Server上创建Mount Point。图15-63 挂载点之下的Segment

可以只在一个或多个或者全部节点上创建某Mount Point。创建Mount Point其实就是在系统虚拟目录下创建一个新路径，创建命令只需要在一个节点上执行，指定所有需要创建Mount Point的节点，即可将同时在对应节点上创建。Mount Point被创建之后，就需要将先前创建好的文件系统挂载到Mount Point中，可以选择只在一个或者多个或者全部节点上挂载某个文件系统到某个Mount Point；同样，命令也只需要在一个节点上执行，但是这个节点必须是管理Root Segment的节点。文件系统可以通过向其中加入新Segment的方式进行空间扩展。如图15-63所示为一个名为TEST-FS的文件系统的底层存储空间由4个Segment组成，第一个Segment为Root Segment，这个文件系统被挂载到了路径/cluster/data1之下。

挂载了文件系统之后，如果需要访问对应数据的应用程序与Segment Server处于同一操作系统内，那么就可以直接读写了。然而，Fusion集群还可以将这个Mount Point以NFS或者CIFS的方式Export出去，可以选择在一台、多台或者全部节点上Export，当然，这些节点必

须已经Mount了对应的文件系统。

Export之后，客户端就可以通过NFS或者CIFS协议来访问对应的目录了。也可以使用Fusion Client来访问。Fusion Client并不直接访问集群，而也是通过下层的NFS Client或者CIFS Client来访问集群。Fusion Client只不过在NFS Client和CIFS Client之上再增加一层控制逻辑以便执行更多功能比如File Allocation等。

Fusion Manager是运行在一个单独PC上的组件，通过前端以太网来与集群所有节点通信，可以用来管理和监控整个集群，但是集群数据IO的过程并不经过Fusion Manager。创建VG、LV、FS、Mount Point等任务可以在Fusion Manager中进行。Fusion Client的Mount Point信息、Mounted Filesystem信息都需要从Fusion Manager中获取，所以对于Fusion Client来讲，Fusion Manager是必需的。

IBRIX已经与Mellanox公司进行了合作，推出了可以使用Infiniband网络来传输NFS协议的集群系统，其后端使用Fusion FS，前端使用NFS Gateway，Gateway使用Mellanox公司提供的NFS/RDMA SDK以实现NFS over Infiniband。实测结果令人兴奋，单个Gateway的吞吐量就可以达到读1400MB/s和写400MB/s。通过增加更多的Gateway，整体系统性能会线性增长。

说明：至此，第10章的最终幻想也被实现，只有想不到，没有做不到，这就是科技。

15.7.2 Panasas和pNFS

Panasas公司是一家生产高性能集群NAS存储系统的厂商。其生产的ActiveStor集群存储系统由于使用了刀片设计，密度非常高，在一个机

箱中可以插满11个刀片，其中每个刀片包含一块1TB的SATA硬盘和一块32GB的SSD硬盘以存放Metadata，加速Metadata的存取。每个刀片最大4GB的内存，这样每个机箱就可以包含10TB的数据存储空间、320GB的Metadata存储空间以及40GB的内存空间。最大10个机箱可以共同加入集群。如图15-64所示为ActiveStor存储系统示意图。

图15-64 ActiveStor示意图

Panasas将自己的集群文件系统称为PanFS。与IBRIX所不同的是，PanFS集群中不是每个节点都可以提供NFS或者CIFS服务端功能的，每个机箱中只有几个特殊的刀片（节点）可以提供Mount Point和Share Directory，客户端只能与这些节点通信才能够挂载相关目录。只有这些节点上维护着整个集群的文件目录映射信息，所以这些节点又被称为Metadata Server，即MDS。整个集群内的所有MDS形成一个小集群，专门负责提供客户端访问入口和维护集群内所有Metadata，以及锁管理。

与IBRIX相同，PanFS也向外提供了三种访问方式：NFS、CIFS、DirectFlow。像IBRIX一样，前两种访问方式下，客户端并不能并行的直接访问集群内其他节点，数据只能从MDS获得，对于不在MDS上存储的数据，MDS会先向数据所被存储的节点请求这份数据然后再将其发送给客户端。而DirectFlow方式就类似于IBRIX中的Fusion Client。其实，DirectFlow其本质是基于Object Storage的pNFS（Parallel NFS，即NFS 4.1），pNFS是一个IETF的标准协议。pNFS的架构其本质与Fusion Client的实现方式相同，即客户端首先从MDS上获取需要访问的目标文件分块所被存储的所有节点的信息，包括地址、分块信息等，然后客户端根据得到的映射信息，并行的向分块所被存储的所有节点发起数据请求将数据取回。

提示： PanFS另一个最大的特点使用对象存储方式。关于对象存储我们在下面章节会有介绍。另一个集群文件系统Lustre的架构与PanFS基本上类似，这里就不做过多介绍了。

15.7.3 此“文件系统”非彼“文件系统”

传统意义上的文件系统，比如FAT16/32、NTFS、EXT2/3、JFS1/2等，都是实实在在的文件系统，也就是说，它们是真正管理着某个文件和底层存储卷上某个Sector或者Block的对应关系的。而所谓“网络文件系统”，它们根本不管理文件与扇区的对应，所以称NFS/CIFS等为一个文件系统，从某种角度来讲有一点歧义，更准确地可以称NFS/CIFS等为“网络文件访问系统”，而NTFS等传统文件系统可以更准确地称为“文件管理系统”。

同样，对于集群并行分布式文件系统或者San文件系统来讲，在Unix平台下它们底层有些则直接使用Ext3文件管理系统来管理文件，而Windows平台则使用NTFS。有些厂商则对EXT3进行了些许修改之后来使用，比如Lustre；有些则选择全部重写一套自己的FS，比如Sotrnext。

在底层文件管理系统之上，增加一层集群分布式文件映射管理系统，外围再包裹上一层NFS/CIFS网络文件访问系统，便成了一个分布式集群并行文件系统了。

如图15-65所示为集群文件系统与操作系统文件系统生态图。

图15-65 底层文件系统与上层集群文件系统的关系

15.7.4 什么是Single Name Space

Single Name Space被翻译为“统一命名空间”。在理解这个名词之前，首先要理解文件系统对外提供的访问方式。我们都知道文件系统在底层管理着上层文件对应底层存储卷或者磁盘上的扇区的情况。在上层，文件系统将文件放到某个目录中，然后目录还可以在目录中。当我们需要访问某个文件的时候，必须首先知道这个文件在哪个目录中，比如Windows下的“D: \data\file.exe”就是一个文件的路径，它表示D分区下的Data目录下的file.exe文件。又例如Unix系统下的“/user/someone/file.exe”。为何Unix不像Windows有C盘、D盘之类的盘符呢？这个问题完全取决于设计方式。Windows默认就是以各个分区为总入口，然后在入口下建立一级一级的目录。而Unix的文件系统则是以全局为入口，各个分区都被“挂载”到某个目录下。比如，可以将分区sda2挂载到“/home/mnt”下，那么我们如果要访问分区sda2中的数据，就需要进入“/home/mnt”，就相当于进入了sda2分区文件系统的根入口。当然，如果用户习惯用Windows，想让Unix文件访问方式与Windows类似，那么可以将分区sda1挂载到“/sda1”，分区sda2挂载到“/sda2”，依此类推。其实在Windows上也可以将分区挂载到某个其他目录下。这种目录叫做虚拟目录，或者Virtual Directory，即目录中存放的并不是原本属于这个目录下的文件或目录，而是另外一个存储空间的目录树。虚拟目录更应该理解为一个路径，这种意义上的路径与目录和文件本身没有直接关联了，路径的唯一意义就是提供一个标记，就像路牌一样，路牌上对应的路名并不等于那条路本身。

在一个多点集群环境中，每个节点都有各自的虚拟目录，或者说路径。然而，集群之所以称为集群，是因为这个集群对外应当表现为一个整体，内部不存在冲突或者重复的事物。比如这个集群对外用NFS Export的某个路径“/cluster/data1”，客户端不管向集群中的哪个节点发起请求使用NFS来Mount这个Export之后，所看到的数据内容都应当是相

同的。这就要求集群内部不会在多个节点上共同存在多份独立的“/cluster/data1”。比如，集群中的某个节点A将自己所管理的某个分区sda1挂载到了“/cluster/data1”路径下面，而这个路径是将被NFS Export出去供客户端访问的，那么这个节点就应当同时通知其他所有节点都生成这条Mount Point，只不过其他节点会感知到这个路径对应的实际存储空间并不位于本地所管理的存储空间，而位于节点A上。那么一旦非A的其他节点接收到针对这个路径的访问请求，就需要将请求发送到A节点执行然后取回结果并返回给客户端（节点后端不共享存储），或者向节点A发起请求将这个路径对应的实体数据空间的映射信息传送过来，然后自己从后端存储空间中读取数据并返回给客户端（节点共享后端存储）。已经被某节点使用的路径，不能再在其他节点上再次挂载，因为会引起冲突，但是其他节点可以将自己所管理的存储空间挂载到“/cluster/data1”的下一级路径比如“/cluster/data1/othernode”中，这样是不冲突的。

在一个非集群环境中，如果有两个NAS Server端，同时存在两个名称相同的路径比如“/cluster/data1”，而且都使用NFS Export出去了，此时客户端挂载的就是两份完全不同的独立存储空间了。集群中所有节点上的供客户端挂载的路径不重复并且所有节点统一协作，统一口径，对外表现一台单一的NAS Server，这就是所谓“Single Name Space”。

15.7.5 Single Filesystem Image与Single Path Image

前面提到过Single Name Space，也就是所谓的全局统一命名空间。这个词的反义词就是非全局多命名空间，也就是说有多个独立的文件系统空间。这两个词都是用于集群文件系统环境中的。

实现单一命名空间有两种方式。第一种是将分布到多个节点上面的

多个独立文件系统进行松绑定。比如将a节点上的/fs/a以及b节点上的/fs/b绑定成同一个/fs下面的两个目录：/fs/a和/fs/b，客户端访问集群中的任何一个节点，比如访问a节点，那么客户端所看到的目录就是/fs/a或者/fs/b，在未实现单一命名空间之前，客户端通过a节点只能看到/fs/a而看不到/fs/b。或者也可以这样搞：两个集群节点各自管理自己的文件系统空间，用一个虚拟化模块将这两个实际的文件系统空间虚拟成一个大空间，比如原来是/a和/b，而虚拟化之后，这两个目录共同融合成了一个/a，/b下面的子目录和文件现在都融合到了/a下面。或者/a和/b下的数据都被虚拟到了一个虚拟目录/c下面。通过这样的简单松耦合方式来实现将多个独立文件系统空间虚拟化融合成一个大嵌套空间的做法，就属于Multiple Filesystem Image，因为这种整合方式并没有影响到各个节点上的本地文件系统，只是在其上层做了一层覆盖虚拟化，只是将目录路径进行了嵌套虚拟，这就必然导致其颗粒度将会非常大，比如某个文件或者某个目录的内容只能被存放在集群中的一个节点上。当客户端通过a挂载/fs目录却发起了对/fs/b下某文件的访问，那么此时a节点会将这个请求通过集群间内部互联网络发送给b节点，b节点处理之后将结果返回给a，然后a再将结果封装返回给客户端。所以说，Multiple Filesystem Image更准确的应该被称为Single Path Image，即单一路径影像。

相对于Single Path Image，Single Filesystem Image则是直接在每个节点的文件系统中作彻底的架构改变。当然，这里所谓“彻底”只是相对的，很多Single Filesystem Image实现方式其实还是在诸如EXT3这样的本地文件系统之上增加一层虚拟化逻辑，只不过这层逻辑与本地文件系统以及其他节点之间结合得更加紧密了，颗粒度大大降低，属于一种紧耦合方式了。比如某个文件虽然在客户端看来是存放在/fs/a下面，但是底层可能是这个文件的前半部分被放在a节点，而后半部分则被存放在b节点中。在Single Filesystem Image中，集群中的所有节点都可以看到整

个集群中的这个大的虚拟的文件系统，并且知道具体哪个目录或者哪个文件的哪个部分存放在集群中的哪个节点，多个节点相当于被同一个文件系统所管理的多个“磁盘”。而Single Path Image模式下的集群，每个节点只能看到和管理它本地所存储的文件，对于其他节点上的文件，更准确地说应该是路径，是靠一个松耦合的虚拟化层简单的嵌套来实现的。

松耦合可以容易地实现更多节点的扩充，紧耦合就不是那么好扩展了，紧耦合需要维护的状态、元数据等信息量都显著提升，随着集群中的节点数量增加，维护和传输这些元数据以及状态就需要更多的计算资源以及网络资源。在实现难度上，Single Path Image基本上没有太大难度，而Single Filesystem Image则实现难度很大。

SPI其实就是多个独立文件系统的松耦合，如果客户端程序从节点a进入，而访问的文件却位于节点b，那么此时a会将收到的IO请求直接通过内部网络通路转发给b，同理，其他节点所收到的针对b节点中文件的访问都会被直接转发给b，这样就相当于b这个独立文件系统像传统方式一样接受外部的IO。而在SFI模式下，接收到客户端IO请求的节点需要将对应的IO进行解析，获得这个IO对应的数据到底落在哪个或者哪些节点上，然后将IO拆分后分别传送到对应的集群节点中进行读取或者写入。

15.7.6 集群中的分布式锁机制

在单个节点的单一操作系统内，存在多个应用程序进程，如果某时刻，应用程序A试图访问一个记录文件F，F中包含有地址簿和电话信息，A将其打开之后，又有一个应用程序B也将F打开，此时，F同时存在于A和B的Buffer内。之后，A将F中的对应某人的电话号码做了更

改，并且将更改写入了F。而此时B的Buffer中的F的内容依然是A做更改之前的，此时B将F中对应这个人的电话号码改成了与A之前改的所不同的值，并且保存到了F中，那么，F的内容就会变为B所更改的，A做的更改将丢失。在Windows系统下，MS Office在打开文件时是对整个文件加锁的，而记事本是不加锁的。大家可以做个实验，打开两个记事本程序，打开同一个文件各自编辑，谁最后一个保存退出，谁做的编辑就被保存到了文件中。我们也可以使用CIFS协议来分别打开一个文本文件和一个Word文件，如图15-66所示，在数据包中，上面为打开TXT文档时程序的行为，可以看到程序并没有对文件加任何锁，甚至允许删除（在本地也是同样的行为，已经打开的TXT文件可以被其他程序删除）。下面为用MS Word程序打开Word文档时的数据包，可以看到程序只允许其他程序读取此文件。

图15-66 Word文件与TXT文件的加锁情况

这显然是一个很大的问题。所以，所有文件系统都会提供一种API，让一个程序在打开一个文件的时候，顺便给这个文件上锁，其他程序不可以打开或者只能以只读方式打开这个文件，只有当加锁的程序退出之后，或者将锁释放了之后，其他程序才可以修改这个文件。即，一旦遇到多个程序需要修改同一个文件，那么这些程序只能排队一个一个的来，如果某个程序给文件加了锁，但是却迟迟不作为，比如操作员离开，而其他操作员终端就只能等待，那么时间就被白白的浪费了。上图中所示的Share Mode只是Windows下提供给程序的一个简易粗粒度锁API，以整个文件为单位设定共享模式，Windows还提供了Lockfile Extend高级锁API（见下文）。

不锁不是，锁也不是，那么有没有两全其美的解决办法呢？当然有。

1. 字节锁（Byte Range Lock）

应用程序可以不要求加锁整个文件，而是只加锁文件中的某段或者某几段字节，这些字节是这个程序当前读入并且打算更改的。而其他程序如果访问这个文件的字节偏移不处于被加锁的范围内，则多个程序就可以并行地读写这个文件的不同部分，互不影响。这样，锁的粒度就被极大的降低了，对应地可以并行读写同一个文件的程序也就可以并行执行了，提高了系统性能。

2. 并行冲突访问锁仲裁

在Byte Range Lock的基础上，如果有多个应用程序同时访问一段字节，或者访问的字节段有交集，那么一次只能够由一个应用程序掌握对这段交集的更改权，其他应用程序可以处于只读状态。

应用程序一般不会被设计为多个进程同时写同一文件的同一段字节，如果真的需要这种应用场景，那么必须加锁—更改—释放，之后由其他程序再加锁—更改—释放，这样才可以保证一致性。

3. 集群中的分布式锁

同样，一个集群就相当于一台大的虚拟的独立系统，集群外的多个客户端就相当于多个应用程序，它们共同并行地访问集群中的资源，如果它们也需要同时并行访问一个文件或者一个文件的各个字节段，那么同样也需要锁机制。如果为了保证一致性，那么程序在打开文件的时候必须显式地对对应字节段加对应权限的锁。然而，由于集群毕竟是由多个节点而组成，那么维护所有的锁这件工作需要由哪个节点负责呢？一般有两种实现方式，第一种是在集群所有节点中选择一个节点专门负责锁

的维护；第二种是所有的节点共同维护锁，锁信息在所有节点上同步。前者称为集中式锁管理，用于非对称式集群；后者则称为分布式锁管理，用于对称式集群。

4. 元数据锁与实际数据锁

在一个共享存储型的对称式集群（见下文）中，所有节点均可以掌管文件系统元数据，所有节点中的元数据信息是完全同步的，一个节点的元数据变化均要通知到其他节点。当某个节点需要为某个文件分配物理存储空间的时候，会锁住相关受影响的元数据，比如空余空间位图。为何要锁住？因为此时只有这个节点知道具体要分配哪些空余块给这个文件，如果不锁住位图，其他节点如果也在分配空余块给其他文件，那么这两个节点所分配的空余块就有可能冲突，导致文件数据被错误覆盖，后果严重。所以这个节点需要利用分布式锁机制来通知其他所有节点，此时由它来操控位图，分配完成之后，将元数据的变化通告给所有其他节点，其他节点同步更新自己的元数据缓存。元数据锁的重要性就体现在这里。而实际数据的锁一般是由应用自己来申请的，集群各个节点不会自己去锁定用户实际文件的某段字节。元数据锁是集群为了保证自身一致性而必须要有的，用户是见不到也调用不到这种锁的。而实际数据锁是文件操作语义层面的。元数据锁与实际数据锁要分清。

15.7.7 集群文件系统的缓存一致性

集群文件系统一般都会有读缓存，即在集群中的每个节点上都会维护一个读缓存，一旦某个客户端应用程序更改了某个节点上存储的内容，而这段内容恰好又与其他节点缓存中的缓存数据有交集，那么有交集节点的缓存中对应的数据就会被作废，不再缓存，或者读入最新数据

继续缓存。利用这种“写即作废”（Invalidate on Write）机制来保证全局缓存一致性。

对于共享存储方式的集群，写入数据的时候最好刷盘，这样其他节点就可以通过读磁盘来看到这些最新的数据。相当于公用物品，用完之后要放回原处别人才可以继续用，而不能先暂存在你这里。某些复杂的系统也可以使用写缓存，对于缓存的脏数据会通知到集群中所有其他节点作废对应的缓存，同时其他节点针对这段数据的操作都与缓存脏数据的节点联系而获取这段数据而不是从磁盘读入，其他节点如果需要更改这段数据，则作废之前的其他节点上对应的这段脏数据，写一次即可。

而对于非共享存储的集群，写入的数据可以缓存在本节点中而无须考虑缓存一致性问题，因为所有其他节点谁想访问这份数据的话就必须从这拿，所以不存在不一致性问题。

对于共享存储型集群，由于底层卷是所有节点共享的，所以除了元数据之外其他实际数据最好都不要缓存，以便将数据实时地体现在底层共享卷中，谁用谁拿。写缓存最好关闭，但是也可以有，一旦写缓存被打开，那么系统需要花费额外的沟通成本来保证缓存一致性了。读缓存一般都是有的。对于非共享存储的集群，读写缓存都有。

如果某集群文件系统是作为一个外服务的存储集群而存在的，那么写缓存最好一律关闭，因为集群节点中并没有类似SAN磁盘阵列一样的电池保护机制，一旦掉电，缓存中的脏数据将丢失。

15.7.8 集群NAS的本质

集群NAS，说白了，就是一网络文件系统RAID 0（或者RAID 10）。如果建立几个独立的NAS Server，客户主机分别Mount这些

Server，然后将不同的数据手动分类存放在不同Server的目录内，这样也可以做到一定的负载均衡能力，其本质与集群NAS要解决的问题相同，只不过在其他方面不如集群NAS系统灵活。统一集群NAS相对于手动Mount多个NFS Server所带来的好处如下。

（1）能做到Single Name Space。比如某个应用程序需要在一个目录下存放几十万个小文件，而如果将这么多文件放在传统非集群NAS上的同一个目录下，其性能往往是非常低下的。此时，需要考虑将这些文件分在NAS中分开不同的目录存放以提升性能。但是这样做的结果是，NAS系统必须对每个目录都Export出来，应用服务器上也必须对每一个Export点进行Mount，所以也会显示为多个目录。这就无法满足应用程序的要求了。你难道可以强行更改这个应用程序，比如，和开发人员商量一下，别让他将这么多的文件都放在一个目录下，能不能分多个目录放？如果是某个系统管理员提出这种要求的话，那就非常不合适和无理了。遇到性能冲突的时候，修改应用程序只能是最后的对应方法。而集群NAS却可以在解决性能问题的基础上又不影响应用，比如将这个文件系统承载于多个Segment上，每个Segment又分配到不同的Segment Server主机上进行管理，而Export的时候只需要Export一个单一Mount Point，应用主机也只需要Mount单一目录，这就很好地解决了问题。

（2）能做到统一管理、故障切换和在线迁移平衡负载。手动部署多个NAS Server，Mount多个目录，手动平衡负载，不但会与应用程序设计造成冲突，而且还有其它诸多不便。比如，如果某个NAS Server出现性能瓶颈，而其他NAS Server的负载却很低，此时只能是望洋兴叹，累的累死，闲的闲死。而对于集群NAS系统来说就大不一样了，可以随时将负载过高的节点上对应的Segment动态迁移到负载低的Segment Server上同时又不影响应用，而且这种迁移根本不涉及到数据移动过程

（Segment Server之间共享后端存储），执行的速度是非常快的，得到的性能提升也是立竿见影的。其次，如果多台独立NAS Server中某台发生故障，那么它所管理的数据也就无法访问了，而集群NAS系统由于后端可以共享存储，所以在某节点发生故障之后，可以由正常的节点接管所有资源，包括后端Segment以及前端接口IP地址，客户端访问继续执行，当然集群NAS也允许任何节点使用非共享存储。但使用非共享存储就不会享受到HA切换和在线迁移Segment所带来的好处了。最后，多台独立的NAS Server已经形成了数据孤岛，不便于数据统一管理，比如Snapshot、Mirror等。

将集群文件系统中的文件用NAS协议输出，这就是集群NAS。使用SPI模式或者SFI模式的集群文件系统均可以输出为集群NAS系统，其表现出来的优劣与SPI和SFI集群文件系统本身的优劣相同。

15.7.9 块级集群与NAS集群的融合猜想

统一存储这个概念是近几年一直在炒的概念，即在同一台存储设备中同时实现FC、ISCSI、NFS和CIFS等存储协议支持，也就是常说的SAN和NAS融合。对于传统的非Scale-Out存储系统，NAS与SAN的融合一般都是通过在SAN设备前加一个NAS机头来实现，NetApp则是完全通过一个机头来实现，其他诸如EMC、HDS等都是采用前一种方式。

面对市场上如此多的集群NAS系统以及集群文件系统，可以说它们早就实现了Scale-Out架构。SAN方面也是从2008年开始由EMC的V-Max以及IBM的XIV两款产品宣布了SAN存储也开始走向Scale-Out架构，其实在此之前3PAR的SAN设备也早就属于Scale-Out架构了，一时间包括Dell Equallogic、HP P4000、Infortrend ESVA等可横向扩展的x86集群SAN存储系统都突然变得被广泛关注，这些产品共同打造出了所谓“新

型高端”存储，开始侵蚀以EMC Symmtrix DMX和HDS USP为代表的传统高端SAN阵列市场。

至此，Scale-Out架构的硬件同时承载了NAS和SAN，它们二者势必也要进行融合，以后，SAN与NAS将会共同被承载于Scale-Out架构的硬件平台之上。

15.8 对象存储系统

对象存储系统（Object Storage System，OSS），或者也叫对象存储设备（Object Storage Device，OSD），本书选用OSD作为这种技术的代名词。

OSD的雏形是由卡耐基梅隆大学的Garth Gibson在1994年提出的，当时被叫做Network Attached Secure Disks（NASD）。几年后，National Storage Industry Consortium（NSIC）对NASD进行了完善和修改，最终被ANSI T10收录成为一个标准项目，命名为T10/1355-D。几年之后，也就是2004年，SNIA组织对这个技术改头换面，成为了ANSI T10 SCSI OSD v1标准，随后又继续对其进行发展，形成了ANSI T10 SCSI OSD v2标准。

那么，OSD到底是何而生的呢？我们都知道NAS协议设备，主机客户端不需要维护文件—块映射，只需要将对文件的操作请求发送到NAS服务端即可得到相应回复，NAS相当于把文件系统逻辑Offload到了主机之外。NAS协议中的NFS协议是Sun公司在1984年左右提出的。10年之后，也就是1994年，卡耐基梅隆大学所发布的NASD也同样使用这种将文件系统底层存储映射管理外置的思想，即把块映射逻辑移出主机之外，放到外部存储设备中执行。但是，与NFS不同的是，NASD的设计之初的构想是把文件系统逻辑直接放置在磁盘中，因为设计者认为磁盘内部处理芯片已经足够强大到可以执行这些高层复杂逻辑了。这样的话，主机直接可以并行地访问每块硬盘来获取文件数据，而不是块数据。如图15-67所示为NASD的系统架构示意图。

图15-67 NASD架构示意图

NASD设计之初就与NFS在架构上有着本质不同，针对同一个Export目录，NFS客户端只能从一个NFS服务节点Mount一次，而且今后所有针对这个目录中文件的IO访问都只能发送给这个NFS服务节点来处理，一个节点的处理能力、网络带宽都是有限的。而NASD打算打破这种限制，NASD可以让客户端主机直接并行地访问所有Disk，所有Disk同时为客户端主机服务，这样，就打破了传统NFS的瓶颈，IO访问数据就可以并行地被处理和传输，性能有很大提升。然而主机客户端如何知道要访问的文件被存储在哪个Disk上呢？难道需要在客户端本地维护一份映射信息么？这样做不就和普通文件系统无异了么？解决这个问题的办法是，NASD在系统中引入了一个独立的Metadata Server（MDS），这台Server上维护着整个系统内的“文件—所在Disk及块”映射关系，任何客户端访问任何文件，都需要先通过网络向MDS进行查询目标文件所被存储的Disk和块列表（某个文件可以以Stripe的形式分散到存储在多个Disk上）；任何客户端打算创建文件或者对文件进行写入，也需要首先告知MDS，由MDS来决定文件将被存储在哪个或者哪些Disk块上，并将信息返回给客户端。在得到列表及其他必要信息之后，客户端主机直接向对应的Disk地址发起数据IO请求来读出或者写入数据。MDS不仅负责元数据管理，而且还负责客户端认证以及文件权限认证，只有通过认证的客户端才能够访问Disk以及对应的文件、目录等。

以上仅为NASD设想中的理想状况，请勿对号入座。我们知道，将文件系统逻辑移到Disk上这个构想，至今仍然只是一个梦。在磁盘上实现高级复杂的逻辑的想法毕竟太过超前与疯狂，甚至比集群化并行访问的思想更加超前，所以我真是佩服Garth Gibson。

我们总结一下NASD的设计架

图15-68 单台主机内部

构：将文件系统逻辑从主机端移出放到网络上的一台MDS上，所有的智能Disk也放到网络上，主机端通过某种NASD客户端程序来向MDS获取元数据信息，并且在得到信息之后根据这些信息直接访问网络上的智能Disk存储文件。我们再仔细思考一下这种架构，这其实和单台主机的内部架构本质上没啥两样。如图15-68所示，在传统的单台主机内部，程序如果打算访问某个文件，文件系统也会查找对应的元数据以获得对应文件所处的Disk以及Disk上的Setcor的位置，查到之后，也会通过磁盘控制器来并行地对多块磁盘进行访问（Raid）以存取数据。它与NASD的架构的差别只在于：文件系统块映射逻辑在主机中运行，Disk位于磁盘控制器之后而不是网络上。但是单台主机所获得的性能可是要远高于NASD，因为文件系统运行在主机内存中，通信也是在内存中；磁盘控制器的主机总线以及控制器连接磁盘的总线速率均远高于外部网络。

虽然NASD的架构设计并不能取代传统主机客户端的文件访问，但是它却可以超越传统的NFS/CIFS架构。

如图15-69所示，左侧的传统NAS访问架构，虽然磁盘数量相同，但是NFS客户端只能够通过一条网络链路来访问NFS服务端节点，而且每份数据都是串行存取的，NFS服务端节点也只有一条链路连接网络；而右侧的NASD架构中，磁盘数量没有变化，但是每个NFS客户端节点却可以直接并行地访问每个数据服务端节点（Disk），并且，随着服务端节点（Disk）的增多，系统性能会线性增长。而如果向传统的NFS服务端中增加再多的磁盘，其所获得性能也没有什么本质提升。

图15-69 NASD与传统NAS架构比较

那么是否传统的NAS也可以改为这种架构呢？当然可以，我们可以手动地将文件分别放入多个节点的多个Export中存放，再在客户端主机

上分别Mount这些NFS Server所输出的目录，然后修改应用程序，访问A文件请走路径a，访问B文件请走路径b。这样也可以达到并行访问的目的，但是由于同一个文件只能放到同一个Server节点，所以不能并行同一个文件；而且这样做也是不可行的，Mount多个节点、修改应用程序，这是完全的强盗逻辑。所以，传统NAS虽然在物理上也可以改为这种架构，但是，由于传统文件系统中的目录、文件、路径等概念，以及NFS所使用的Mount Point的做法，却注定它不能够实现对同一文件的并行。NASD并没有Mount的概念，它对“文件”和“目录”等抽象概念进行重新定义和封装（详见下文），正因如此，NASD才可以做到对任何要访问对象的并行操作。

至此我们明白了NASD的价值所在。Disk这个词是不是可以在NASD的概念中消失了呢？因为至今也没有实现智能Disk。所以下文中不再用Disk，而将Disk改为OSD，即Object Storage Device。这里何谓“Object”呢？如图15-70所示为OSD层在系统IO路径中的示意图。图中右侧下部即为一个OSD。左侧则为传统Block级访问模型。

图15-70 OSD协议在系统IO路径中的位置和作用方式

可以看到，OSD在协议层面与NAS协议本质上是相同的。它之所以被称为“Object Storage Device”，是因为在OSD的概念中，文件不叫File了，叫Object；目录也不叫Directory了，叫Partition。在此之上，OSD相比NAS增加了一些改进，比如安全认证等方面。其次，OSD中每个Object都用一个128b的Object ID（OID）表示，说白了这个OID也就对应了NAS协议中的File Handle。

如图15-71所示为OSD中的各种主要概念。像普通文件系统一样，OSD设备一样需要维护每个文件，哦，应该说是Object的块映射、元数

据、属性和OID等信息。Object分为多种类型，每种类型的Object含义如下。

图15-71 OSD中的各种概念

- Root Object: 表示OSD设备本身。
- User Object : 被客户端主机所创建的Object。
- Collection Object: 一组按照某种条件而聚集起来的Object组合，比如“*.exe”，即“所有exe文件的集合”就是一种Collection Object。
- Partition Object: 一个用来盛放User Object的容器，相当于目录。

OSD不可能由一个磁盘来充当了，那么谁来充当呢？当然只能用一台独立的服务器来充当。这台服务器与NFS服务器底层处理方式类似，都是维护一个本地文件系统，管理Object与底层磁盘块的映射关系以及其他元数据等信息，对外使用一种专门的协议来供客户端访问其保存的Object。

可以说NASD是一种极其超前的思想和技术。它早就将集群环境考虑在内，可惜的是，20世纪80年代，集群并未得到发展，所以NASD也并没有成为潮流技术，而被随后出现的NFS和SMB协议超过了。

但是NASD却一直没有停止发展，一直到被SNIA推进并最终在2004年被收入T10，成为一个标准，也就变为现在俗称的OSD了。

有人可能会问：T10不是专门搞SCSI协议的组织么？怎么开始搞起文件级访问协议了呢？说对了。也正是因为T10只搞SCSI，所以才会收录它，因为，OSD就是使用SCSI CDB来传输针对文件的IO请求的。对

于当初NASD是否也是使用SCSI，不得而知，但是SNIA提交到T10的OSD协议确实是使用了SCSI CDB来描述文件级的请求。如图15-72所示为OSD协议在SCSI协议层次中所处的位置。

图15-72 OSD协议在SCSI协议层次中所处的位置

思考： 其实读者此时应该已经早就了解了所谓的“IO三大件”了，每个IO，不管是标准协议，还是私有函数调用，基本上都是由“目标”、“起始地址”和“长度”这三大件组成的。对于块IO，这三大件就是目标LUN ID、起始LBA地址、要IO的LBA地址长度；而对于文件IO，这三大件就是文件名、起始字节地址、要IO的字节长度。可以看到文件和块这两种IO方式，其本质是一样的，本来文件底层也就是对应的数据块，文件系统只不过做了一层地址翻译封装和组织。那么这两种IO从本质上来讲就可以被融合。对象存储系统从某种意义上讲结合了块级访问的高效与文件访问的灵活性和便捷性。

OSD将文件访问协议也统一到了SCSI协议集中，做到了Block和文件级访问的大统一。NASD所做的最大贡献其实并不是Object，也不是后来被标准化的OSD协议，而是其针对集群环境下文件的高性能访问所作的设计，这个思想最终催生了一大批有着类似架构的集群文件系统，包括Lustre、PolyServe、Ibrix、PanFS等。

这些集群文件系统拥有共同的基础架构，如图15-73所示为这种架构的示意图。

图15-73 基于OSD的集群文件系统普遍架构

它们有些使用OSD来作为访问协议，也有些使用私有的但是与OSD

类似的协议。目前几乎所有的集群文件系统都同时提供了基于NAS（CIFS/NFS）的访问接口。客户端主机可以选择使用OSD或者NAS方式来访问集群。至于具体的访问方法，请参考本章其他部分，这里就不做过多介绍了。

部署OSD，不但需要OSD存储设备，还需要在客户端主机上安装OSD Initiator，就像NFS或CIFS需要使用对应的NFS Client模块和SMB模块一样。Panasas的 DirectFlow就是一种OSD Initiator。

用户程序本身无须做任何更改即可使用OSD作为存储设备。用户可以挂载一个由MDS虚拟出来的路径，当在这个路径下创建一个文件或者访问这个路径中的某个文件时，OSD Initiator会与操作系统内核的目录虚拟层（比如Linux下的VFS层）进行对接，将用户程序针对VFS路径下被Mount的OSD路径下某文件的访问，翻译封装为相应的OSD SCSI CDB并发向OSD设备以执行这个IO请求，当然，之前应当先去MDS查询待访问目标所被存储的OSD设备和块列表。OSD Initiator向OSD发送的请求中只需要给出Object ID即可，而无须像传统NAS协议一样给出这个Object的绝对路径。OSD设备会自行查找这个OID所对应的底层块并做相应操作。

OSD目前被使用的非常少。根据一些极力推广OSD的厂商的态度，他们认为传统的文件系统，不管是从底层的元数据组织方式上，还是上层的访问接口方面来讲，已经不能够满足海量文件的存储和访问了。而OSD却能够从根本上解决这个问题。

集群文件系统提供的NFS或者CIFS访问接口，客户端主机只能从集群中的某台节点上来Mount某个Export输出目录，而所有的数据IO也都是通过这个节点进行，这也就从根本上限制住了传统NAS协议访问集

群所能够获得的最大性能。而OSD从一开始就考虑到了这种限制，OSD冲破了这种限制。所有使用OSD协议访问集群的客户端主机，它们都首先从Metadata Server上查询待访问的目标文件/Object都分布在哪些OSD节点上，得到所需的信息之后，客户端主机的OSD Initiator会同时并行地向所有存储有这个目标文件碎片的OSD节点发起IO请求，并行地存取数据，这就比传统的NAS协议有了质的飞跃。

思考： 我们观察一下图15-73，可以发现一些玄妙的东西。即，在这整个集群内，OSD相当于Disk，MDS相当于FS，每个Client主机相当于应用程序，以太网交换机相当于系统总线。我们仔细思考一下，这整个集群，是不是就像一台单独的计算机一样呢？多个应用程序同时通过文件系统来访问磁盘上的数据。

而我们再仔细观察一下图15-69左侧的传统NAS架构，再观察一下右侧的NASD架构。你发现这其中的奥妙了么？是的。NASD相当于把上面所述的“系统总线”扩充了，由单条串行改为多条并行；还有，NASD相当于把底层做了Raid0了。

这就是“轮回”二字的奥妙所在。

传统NAS也不甘示弱，NFS的最新升级版NFS 4.1，即pNFS（Parallel NFS），也采用了这种集群架构以及OSD访问协议标准。而CIFS/SMB协议的始祖Microsoft在其Windows Vista/Server 2008/7操作系统中提供了SMB 1.0的升级换代版——SMB 2.0，其在单机访问时相比SMB 1.0有了很大的效率和速度提升，而使用MS提供的DFS也可以达到与集群FS类似的效果，但并不是那么纯粹。传统NAS与OSD将会走向融合。

15.9 当前主流的集群文件系统架构分类与对比

存储系统中的集群系统可以分为两大部分，一个是像V-Max、XIV、InserveT这样的集群SAN系统，它们的集群化对客户端来说是完全透明的；第二种就是集群文件系统，集群文件系统又可以分为其他多种类别。下面我们从不同的角度分析一下目前的集群文件系统的分类。

15.9.1 共享与非共享存储型集群

如果某个集群中的所有节点是共享使用后端存储的（这里的共享不是说共享一台磁盘阵列，而是共享访问同一个或者多个LUN），那么这个集群就属于共享存储型集群，否则便是非共享存储型集群。如图15-74所示，左侧为共享存储型集群示意图，右侧为非共享存储型。但是不要被图中所示的场景所误导，非共享存储型集群不一定每个节点都必须用自己本地的磁盘，节点当然也可以连接到一台或者几台磁盘阵列中来获取各自的存储空间，但是各自的存储空间只能自己访问，其他节点不可访问，这就是非共享的意义。

图15-74 共享存储与非共享存储型集群架构

共享与非共享存储型集群对比如下。

（1）非共享模式的集群文件系统，当某节点需要访问其他节点上的数据时，这些数据需要在前端交换机中（一般是以太网）传输，速度偏慢；而共享存储型则每个节点可以直接对后端存储设备对应的LUN进行读写，在前端传输的只有集群间的元数据沟通流量而不是实际数据流量。

(2) 缓存一致性，共享式需要考虑，非共享式不需要考虑。

(3) 对于非共享式集群文件系统，为了防止单点故障，需要将每个节点上的数据镜像一份存在其他节点；而共享式集群，一个节点故障，另外的节点可以同时接管前端和后端，因为后端存储是所有节点共享访问的。

(4) 非共享式集群可以不使用SAN阵列，服务器节点本地槽位多的话使用本地磁盘也可以满足大部分需求，不可以使用DAS磁盘箱等；共享存储型集群则必须使用SAN阵列。

非共享存储型文件系统又可被称为“分布式文件系统”，即数据被分布存放在集群中多个节点之上。

15.9.2 对称式与非对称式集群

如图15-75所示为对称式集群示意图。所谓对称式集群文件系统是指集群中所有节点的角色和任务都相同，完全等价。在对称式集群文件系统中，每个节点都很“聪明”，它们每时每刻都能够保持精确的沟通与合作，共同掌管着全局文件系统的元数据，每个节点要更新某元数据时都会先锁住，这样其他节点就必须等待，就这样轮值执行任务，保证了文件系统元数据的一致性，同时也精确地保持着缓存一致性。各个节点间沟通量很大。

图15-75 对称式集群文件系统架构

如图15-76所示为非对称式集群示意图。在非对称式集群中，只有少数节点是“聪明”的，其余都是傻节点。也就是说，只有少数节点（一般为两个主备关系的节点）掌管着系统内全局的文件系统信息，其他节

点均不清楚。当其他节点需要访问某文件时，需要首先联系这个聪明节点，后者将前者要访问的文件所对应的具体信息（比如存放在后端哪个LUN的哪段地址，或者存放在哪个节点中）告诉前者，前者得到这些信息之后便直接从后端的LUN或者对应节点中访问该数据。每个傻节点上安装一个代理客户端程序来与聪明节点通信。图中可以看到一些具体的通信过程。这个“聪明”节点叫做“Metadata Server”，简称MDS或者MDC（Metadata Controller）。MDS是系统中唯一掌握文件系统元数据的角色，下文中还会有涉及MDS的描述，请注意阅读和关联理解。

图15-76 非对称式集群文件系统架构

对称式集群文件系统的典型代表有Veritas ClusterFS、HP Ibrx。非对称式集群文件系统的典型代表有Stornext SNFS、EMC MPFS、IBM SanFS。显然，由于第二种方式易于实现，集群间沟通成本低，所以对应的产品也多。对于对称式集群，客户端可以通过挂载任何一个节点上的目录输出即可访问到集群中所有数据，或串行或并行；但是对于非对称式集群，客户端只能通过MDS节点来挂载，一定程度上造成MDS节点瓶颈。但是由于对称式集群的沟通复杂度太高，不利于扩展到太多节点数量；而非对称式集群可以通过引入多个MDS节点来均摊负载。在节点数量较少，也就是在对称式集群的可容忍范围之内时，对称式集群由于每个节点都可以充当非对称式集群中MDS的角色，所以往往能够表现出更好的性能。

15.9.3 自助型与服务型集群

上面图示中的拓扑结构均为“自助型”文件系统集群。何谓“自助型”，顾名思义，也就是谁用谁就形成集群，自我集群，自种自收。比如有个视频编辑应用集群，100台PC Server，每个节点上装有视频编辑

程序，形成了应用集群，现在这些节点想使用一种集群文件系统来共享地访问系统内的所有文件，那么就可以在这些应用节点上直接安装部署集群文件系统，应用集群同时也变成了文件系统集群，每个节点既是形成集群的服务者，同时还是数据的生产者（数据由其上的应用程序生产），同时又是直接访问底层文件数据的消费者，这就是所谓“自助型”的含义。

那么再来看看图15-77所示的拓扑架构，框内的是一个集群文件系统，框外又增加了一排客户端主机，左右两侧分别为对称式和非对称式集群。在这个拓扑中，集群内的服务节点自身并不是数据的消费者，而只是服务者、提供者，而集群之外的客户端主机通过某种访问协议来访问集群内的文件数据。这就是所谓“服务型集群文件系统”。

图15-77 服务型集群文件系统架构

为何会出现服务型文件系统集群呢？自助型不是很好么？还节约了服务器主机的数量。究其原因主要有如下两个。

- 降低成本。自助型集群中每个节点均需要高速IO适配器比如FC来访问阵列存储空间，随着集群规模扩大，适配卡、交换机、线缆等的成本不断攀升。
- 可以接入更多的客户端。服务型集群可以用较少的集群节点服务于较多的客户端主机。集群内部的沟通成本可以控制，实现高速高效。同时外部客户端之间不需要互相沟通，所以客户端数量可以大幅增加。对于自助型集群，如果节点数量太多的话，集群内部沟通信息量以及复杂度将会成几何数量上升，不利于扩充。

15.9.4 SPI与SFI型集群

为了实现集群所必须实现的Single Name Space，有两种方式，如图15-78所示。

图15-78 SFI与SPI集群文件系统架构

- 懒人做法：既然每个节点上都有各自的文件系统，我就把他们输出的路径虚拟化一下，倒一下手，集中管理起来，然后再次向外输出成一个Single Path Image（SPI）。我只管路径统一，不管文件放在哪里。典型代表：微软的DFS。
- 勤快人做法：每个节点都知道所有文件的位置，在文件系统底层进行整合而不是表层的路径整合，即Single Filesystem Image（SFI）。典型代表：CFS等大多数集群FS。

一分耕耘一分收获，SFI可以做到将一个文件切开分别存放到各个节点中；而SPI无法做到。但有时也不得不服投机取巧的效果：SFI往往扩展能力有限，而SPI则可以整合大量的路径（节点）。

因为对于SFI模式的集群FS，其节点之间需要时刻同步各种复杂的状态，每个节点所维护的状态机非常复杂，同步这些状态需要不少通信量，但是外部网络速度永远比不上内存速度，所以这些通信会增加每个节点状态机变化的延迟，导致处理速度有所降低。尤其是当节点数量增多时，比如几十个甚至上百个，那么对于SFI模型的汲取FS来讲就是梦魇了，几十个状态机之间的相互协作，加上外部网络带来的延迟，此时所有这些劣势将会加成，可能导致性能不升反降。而SPI模式的集群FS就没有这个问题，节点之间相互独立，所以需要同步的信息很少，如果主机端不使用特殊客户端访问，而只是通过传统的NFS或者CIFS等访问的话，那么集群节点间可能会出现实际数据的交换，此时就需要一个高

速的内部交换矩阵才能获得较高的性能。

15.9.5 串行与并行集群

对于服务型集群，客户端可以通过两种方式来访问这个集群所提供的的数据：第一种是串行方式，即客户端通过挂载集群中某个节点所输出的目录，之后所有的通信过程都通过这个节点执行；第二种则是并行访问方式，首先客户端初始时也是通过集群中的某个节点挂载对应的输出目录，但是挂载之后，客户端只通过这个节点来获取待访问文件的元数据信息（见图15-76所示的过程），得到文件对应的块地址等信息之后，客户端可以直接利用所获得的信息访问集群中的其他节点来访问对应的数据，如果某两个文件分别存放在集群中不同的节点，或者某个文件被分散存放在多个节点中，那么客户端可以并行地访问这多个节点从而并行地读写对应的文件。

图15-79所示为对称/非对称式两种模式并行访问文件系统集群架构示意图。如图15-80所示为一个非对称式集群并行访问过程中的示意图。图中的OID表示对象存储协议中的Object ID。对于并行访问集群来讲，客户端一般是采用对象存储协议来访问集群中的数据节点的（比如Lustre、Panasas等），当然，也有依然采用NFS方式来并行访问集群中数据节点的（比如Ibrix的Fusion Client）。

图15-79 两种并行访问模式

图15-80 并行访问集群时的步骤示意图

并行访问集群与串行访问集群对比如下。

- 在单条链路速率相同的情况下，并行永远强于串行。

- 目前前端客户端一般都是用1GbE以太网来访问集群，如果是串行方式，客户端只能与一个节点通过这条链路通信，如果这个节点的处理能力不足或者本地带宽饱和，那么对应的客户端所获得的数据带宽也就会受限，可能连1Gb带宽都远未达到。为此，让这个客户端并行地与多个节点通信来访问数据，则可以最大程度地饱和链路带宽。实际测试显示，10GbE的链路下，单条NFS流远远无法满足带宽，最差时可能只有10%的带宽能够利用。这也是之后pNFS并行访问协议被引入的原因。

提供并行访问能力的集群典型代表：Ibrix、EMC MPFS、Lustre、Panasas。基本上所有的集群文件系统都提供并行访问客户端。

15.9.6 集群/并行/分布式/共享文件系统各论

大家平时可能听到过多种叫法：集群文件系统、San共享文件系统、分布式文件系统、并行文件系统是四种主流的叫法，那么这些概念之间到底有什么联系呢？

- San共享式文件系统：其实这种叫法狭义上指的就是自助型、共享存储型的集群文件系统。广义上则也可以泛指共享存储型的集群文件系统，可以是自助型，也可以是服务型。但是最常用的还是诸如Stornext和IBM SanFS这样的自助型共享存储集群。San共享文件系统又可被简称为“San文件系统”。Stornext的SNFS共享文件系统最新版本已经发布了Distributed Lan Client（DLC），从自助型转向了服务型，DLC可以让集群外的客户端并行地访问集群节点，也就是并行文件系统所提供的并行访问客户端代理程序。
- 分布式文件系统：同一个文件系统下的文件（或者同一个文件的多个部分）不是被放在单一节点内，而是被分开存放在多个节点

之内，这就是所谓“分布式”的意义。分布式与共享式是对立的，所以分布式文件系统等价于非共享存储型的集群文件系统。

- 并行文件系统：可以提供并行访问的集群文件系统。客户端访问这些被分开存放的文件时，可以直接从多个节点并行地读取多个文件，或者一个文件的多个部分，也就是并发地直接从存有对应数据的节点上来读写这些数据，这就是所谓“并行”。相对于并行的是串行，串行文件系统，就是指客户端只能从所有节点中的一个节点来读写所有数据，如果需要读写的数据不在所连接的节点上，那么需要由这个节点来向存有对应数据的节点发起请求，将数据从对应的节点通过内部交换矩阵传输过来之后，再传递给客户端。也就是说数据是串行地传输的。分布不一定并行，但是并行一定是分布的。并行文件系统均需要在主机客户端安装一个代理，或者一个新的文件系统挂载器，用来专门实现并行访问。
- 集群文件系统：分布式文件系统、并行文件系统、共享式文件系统，三者统称集群文件系统。“分布式”和“共享式”指的是集群中数据分布的方式，而“并行”指的是用户对这些数据的访问方式。分布、访问，两个层面，两种含义。

15.9.7 集群NAS系统的三层架构

这里再提一下集群NAS。一个集群NAS系统其实可以被分为三层架构，第一层是底层存储空间层，这一层可以是Share Everything（共享型）或者Share Nothing（非共享型）两种模式；第二层是集群FS层，集群FS层建立在底层任何一种模式（共享或者非共享型）的存储空间之上，这一层可以做成Single Path Image和Single Filesystem Image。对应第一层，第一层模式为Share Everything的一般都在第二层使用Single Filesystem Image模式；而第一层使用Share Nothing模式的，第二层既可

以使用Single Path Image模式，也可以使用Single Filesystem Image模式。第三层就是NAS协议输出层了，这一层有四种访问模式：传统CIFS、传统NFS、pNFS并行客户端以及私有客户端。后两者可以并行访问，前两者只能串行访问。

15.9.8 实际中的各种集群拓扑一览

上面介绍了5大类共10种不同角度的集群文件系统架构，利用这10种不同的方式，可以两两组合成任意模式的集群文件系统。下面列一下实际中主要的集群FS拓扑。

1. 直连后端FC/ISCSI SAN

这种形式的集群文件系统就是上文所述的SAN共享文件系统。客户端代理程序在查询MDC得到信息后直接访问后端的SAN磁盘阵列，查询MDC的时候走前端以太网，执行实际IO请求的时候走后端FC网，IO请求遵循SCSI3 SBC纯磁盘级块协议，如图15-81所示。这种模式的集群文件系统典型代表有：Stornext SNFS、EMC MPFS、IBM SanFS。

图15-81 直连后端FC/ISCSI SAN

2. 引入IO节点

在上图中再插入一层处理模块，将后端的LUN虚拟化管理起来，然后再通过另外一种形式进行输出从而被客户端以另外的方式访问，那么系统的架构如图15-82所示。如果在LUN上层引入一种对象存储网关设备，那么客户端就需要使用支持OSD协议的客户端代理，比如pNFS客户端；如果引入一种厂商自行开发的私有设备，比如Ibrix的Segment

Server，那么客户端也需要安装相应的代理程序；如果被引入的是一个NAS头，那么客户端只需要使用标准的NFS或者CIFS客户端即可。至于引入IO节点的原因和优势，请参考后文的论述。这种模式的典型代表有：Panasas、Lustre、Ibrix。其中Panasas使用标准的OSD节点以及pNFS协议；Lustre则也是用OSD节点，但是客户端并非使用标准pNFS协议访问，而是一种私有协议；Ibrix则使用独自设计的IO节点，客户端代理也是使用私有协议访问IO节点。

图15-82 引入IO节点之后

3. 用OSD作为IO节点

同上，其结构如图15-83所示。典型代表：Panasas、Lustre。

图15-83 用OSD作为IO节点

4. 使用标准pNFS协议访问IO节点

见上文，其结构如图15-84所示。典型代表：Panasas。（Panasas的IO节点并非连接SAN后端的LUN，而是使用自己本地的磁盘。）

图15-84 用pNFS协议访问IO节点

5. 使用标准NFS/CIFS协议访问IO节点

如图15-85所示为一个Isilon的OneFS架构简图。其实并不复杂，Isilon使用多个NAS头联合起来组成一个集群NAS系统，每个NAS头都是用自己本地磁盘作为存储空间，所有NAS头的角色都相同。整个系统可以做到Single Name Space，访问任何一台NAS头，就可以访问到全局

的数据。如果待访问的目标数据不在所连接的NAS头管辖范围内，那么对应的NAS头会通过后端的以太网或者IB网络从管辖对应数据的那个NAS头上将数据取过来然后返回给客户端，写也是类似动作。这种架构本身并没有对单客户端或者单文件访问的速度有任何加成作用，但是它却做到了所有节点平等化以及真正的全局命名空间。

图15-85 使用标准NFS/CIFS协议访问IO节点

6. 所有节点角色一致的集群系统

见图15-61。典型代表：**Ibrix**、**Isilon**。这种模式下，集群中所有的节点既充当IO节点，又充当MDC控制器，访问任何一个节点均可访问到全局数据。

7. 所有IO节点均使用本地磁盘的集群系统

见图15-64。典型代表：**Isilon**、**Panasas**。**Isilon**使用机架式服务器，**Panasas**使用刀片服务器。这种模式下，IO节点并不连接后端第三方磁盘阵列来获取存储空间，而是使用每个刀片或者机架中本地的磁盘，利用大量的节点来获取高容量存储空间。

综上所述，集群文件系统其实最后演化为了两大阵营：一个是客户端直接访问后端SAN的模式，另一个则是在客户端和后端FC SAN LUN之间引入基于以太网链路访问的IO节点的模式。后者又可以根据客户端访问IO节点使用协议的不同而分为更多的种类。

两大阵营各有利弊。直接访问后端SAN的模式下，客户端与后端的磁盘阵列之间没有任何其他处理模块，所以其IO的效率是最高的，而且

加上FC网络的速度，整个系统的速度和效率均较高。但是相对来讲，其成本也将随着客户端数量的增大而正比增加，因为目前FC适配卡的价格依然居高不下，如果为每个客户端安装一块或者两块FC卡，也是一笔不小的投资。此外，由于后端的LUN皆由MDC来挂载和管理，而系统中的MDC数量有限（目前最多两个），所以一旦两个MDC都出问题，那么整个系统就瘫痪了。引入IO节点之后，一方面客户端可以使用廉价的以太网来访问IO节点了，花费降低；另一方面，对于像Ibrix这种架构，所有节点都同时作为MDC和IO节点，IO节点本身可以共享访问后端所有的LUN，所以一旦某个IO节点故障，那么其他任何一个IO节点就可以接管故障节点之前所挂载的LUN以及文件系统，继续提供服务，只要系统中还剩下一个IO节点/MDC，那么整个系统就不会瘫痪，也就是说，这种模式下的系统容错率高了很多。但是，随之而来的问题就是IO效率相对低了，以及客户端IO速度的限制，以太网毕竟只有1Gb/s的速度，在单客户端或者单文件访问的情况下，集群系统显示不出多少优势，但是直接访问后端SAN的模式下，不管是单客户端还是单文件访问，其依然能够达到较高的速度。

提示： 关于单客户端和单文件访问的问题，下文会有更详细的论述。

15.10 带外共享SAN文件系统

在广电领域经常会用到一种架构，就是带外共享文件系统。这种架构允许多台主机共享访问同一个或者一批Lun，但是必须在一个特殊的文件系统的协调下进行，这个文件系统掌握全局的文件系统元数据，每台主机虽然可以直接读写Lun，但是在读写之前必须询问和经过这个文件系统的同意，文件系统会告诉主机应该怎么访问以及访问哪些地方。

15.10.1 SAN共享文件系统

多主机能否同时读写同一个LUN中的文件而同时保证数据一致性，一直是很多人在反复问的问题，答案也是固定的，即必须通过使用特定的共享式文件系统来实现。即便所有客户端主机同时使用DIO+WriteThrough模式的IO方式来访问对应的文件，由于文件的Metadata始终是被缓存在所有客户端节点的，而DIO和WT是不能把Metadata也同步刷入磁盘的，所以这种做法只适用于Metadata永不改变的情况下，而且性能也得不到保证。

这个问题的本质原因就是整个系统内存在多个独立的文件系统逻辑，即每个客户端都要维护自己的文件系统缓存，而它们之间又互不通信，各做各的。那么很显然，解决这个问题的根本办法，就是让整个系统内只存在一份文件系统逻辑和缓存。这种做法的一个例子就是NFS或者CIFS等NAS协议访问方式，全局的文件都由一台单一的NAS服务器来处理，客户端不需要处理文件系统逻辑。这样的话，只要客户端应用使用同步IO调用，即可完全保证数据一致性了。

至此，多主机共享访问同一个LUN下面的文件的问题，有了一种解

决办法，即使用NAS。然而，还有另外一种办法，即保持客户端的访问协议和底层链路速度不变，直接把多余的文件系统实例去除掉，只保留一份文件系统逻辑。这种思想也就演变为了共享式SAN文件系统了。

如图15-86所示，左侧为整个系统原先的架构，多个主机共同访问同一个LUN中的文件，会产生数据不一致，因为每台主机都各自为政，当任意一台客户端打算访问目标文件的时候，它们各自会向自身内存中的文件系统发起请求，从而执行IO操作。

图15-86 传统文件系统与SAN文件系统比较

右边所示的架构则为进化之后的架构，5台客户端中的4台上的文件系统管理逻辑被剔除（所谓被剔除，并不是说将操作系统中的文件系统去掉，而是引入一个可挂载的新文件系统），只在其中一个客户端主机上保留了一份。当其他4个客户端需要访问文件的时候，由于对应的文件系统已经不在本地了，而位于第5个客户端之上，所以它们需要通过网络来向位于第5个客户端上的文件系统发起请求（由一个代理程序模块发起），也就是图示的以太网。5号客户端上的文件系统接收到查询请求之后，将对应的信息，比如待访问部分所对应的LUN以及LUN中的LBA地址段等发送回其他客户端，发起查询请求的客户端收到这些数据之后，需要自己来从存储设备中读出或者写入对应的数据块。

经过这样的改造，就可以避免同时访问时造成的数据不一致状况。而运行唯一文件系统管理模块的那个客户端的身份也进化了，我们称它为“Metadata Server”或者“Metadata Controller”，即MDS或者MDC。MDS如此重要，所以一般都会使用两个或者多个MDS组成一个HA的Failover组以实现容错，有些甚至可以实现双Active负载均衡。

至此，还有一点比较重要的东西需要理解，即在图15-86左侧所示的传统方式下，数据的查询和数据的IO都是由同一个客户端执行的，而右侧的架构中查询由MDS上的中央文件系统来执行，而数据IO则需要每个客户端自行执行（直接访问后端存储设备，无须经由MDS）。这一点与单一NAS服务器不同，SAN文件系统属于一种带外架构，而单一NAS服务器属于带内架构。理解这一点，是理解后面并行访问集群化进化的关键。

15.10.2 针对NAS和SAN文件系统的并行化改造

由于传统NAS系统属于带内架构，所以NAS前端的有限以太网链路以及单一NAS服务器后端的带宽限制最终制约了单一NAS服务器的性能，为了解决这个问题，集群NAS以及对应的并行访问协议比如pNFS等被开发了出来，NAS领域也出现了带外MDS的概念。客户端使用pNFS Client向MDS查询待访问目标的位置等信息，得到信息之后，自行向目标发起访问。与单一NAS服务器架构不同的是，集群NAS系统中存在多个存储节点，客户端可以同时从所有这些存储节点中读写数据，这就是所谓的“并行访问”了，所以也就是一个集群。pNFS使用了OSD来作为客户端与存储设备之间交互的协议。

同样，SAN文件系统本身就是一个带外架构，它实现并行集群化访问是原生就支持的，只要在系统内添加更多的存储阵列、更多的LUN即可。如图15-87所示为集群化并行SAN文件系统和集群化并行NAS系统的示意图。

图15-87 集群化并行系统示意图

我们分析到这里就可以发现，其实集群NAS和SAN共享文件系统这两者的本质是一样的，殊途同归，最终两者都使用了MDS作为中央单一文件系统管理者，各个客户端使用对应的代理程序（比如pNFS客户端、SAN文件系统客户端）来直接并行地访问存储设备。只不过在进化的过程中，SAN文件系统是一步到位，而NAS则经历了比较复杂的演化。

集群系统的再进一步进化的形态，就是类似Ibrix这样的，任何一个节点都可以充当MDS，使得集群中的每个节点都具有相同的地位。

经过集群化改造之后，整个系统变成了彻底的Scale-Out架构，随着系统内部存储节点的增加，客户端所获得的带宽和IOPS也就可以线性上升。

15.10.3 SAN FS实例分析

下面我们就来看一下一个SAN文件系统的底层架构吧。

1. 核心本地文件系统EXFS

如图15-88所示为该SAN FS的底层架构简图。其中MDC上的核心基础模块为一个被命名为EXFS文件系统，EXFS是一个Linux上的纯本地文件系统，由于EXT3存在诸多限制，所以这个厂商自己写了一个全新的EXFS文件系统。EXFS不仅具有文件系统的基本功能，而且还负责卷的管理和挂载，相当于一个集成了卷管理模块的文件系统，支持管理底层最多4096个LUN，而且支持在线扩容。可以在线向对应的文件系统内添加LUN，当用户需要从某个文件系统内剔除某个LUN的时候，EXFS还提供了对应的工具来检测哪些文件处于这个LUN中，从而可以让用户

手动地将文件迁移到别处，然后再剔除对应的LUN。EXFS将整个文件系统的元数据存放到一个或者多个专属的后端卷中存放，而不是遍布后端所有LUN，这样做的好处是充分保证性能和安全性。

图15-88 该SAN FS底层架构图

此外，EXFS的设计在文件分布策略方面颇有考究。

提示： EXFS对于文件的分布给出了三种方式。第一种就是完全条带化分布方式，相当于一个盲RAID 0模式，每个具有一定尺寸的文件都被拆分多份放在后端的多个LUN中。这种模式对于单个大文件的单线程访问具有最好的性能加成。

第二种则是纵向分布模式。在本书的“IO路径及优化”相关章节中曾经详细分析过盲RAID 0所存在的问题，即多文件/LUN并发的时候，性能骤降。而这对于数字媒体系统是致命的，会直接导致视频播放的卡壳。针对于此，EXFS可以实现将一个文件存放在一个LUN中，下一个文件存放在下一个LUN中，以此类推。这样就可以保证并发访问多个文件的时候，不会产生LUN的IO冲突，当然这些LUN在底层也最好要位于不同的磁盘组中，最终IO冲突的是磁盘而不是LUN。

第三种则是为了配合MAID技术而生的，MAID可以让长期没有IO访问的磁盘的盘片停止旋转从而节电。EXFS在存储文件的时候可以将文件先存满一个LUN，然后再存储到下一个LUN，以此类推。这样，其他没有存储文件的LUN所对应的磁盘就可以停止旋转了。但是这种分布方式会影响性能。

用户可以根据自己的需求来选择不同的分布模式。

2. MDC上的信息传递员——NFS Daemon和Layout Daemon

有了EXFS这个核心之后，必须还要有一个负责将文件系统元数据信息传递给客户端上的代理程序的角色。很显然，这种角色早就存在了，比如NFS网络文件系统，客户端可以通过一系列的RPC Call（比如GETATTR()、FSINFO()、FSSTAT()等）来向服务端查询文件的各种属性。该文件系统并没有沿用传统的NFS服务端式的带内模式，而是在它上面进行了改进，增加了一个新层次来实现更多的功能（见下文）。从这个角度上来看，MDC相当于一台带外的NFS服务器。

提示： Ibrix以及其他多个集群FS厂商的MDC节点其实也是使用NFS Daemon来负责接收客户端的查询请求，同样，Ibrix Fusion Client的底层也必须依靠NFS Client。不同的是，Ibrix在Windows上直接使用了微软提供的SFU/SUA来作为NFS Client。

这种特制的服务端与传统的NFS服务器有一点最大不同，那就是客户端针对文件的实际数据IO操作（对应着NFS协议中的READ()、WRITE()等）并不发送给MDC来处理，因为如果连实际IO都要MDC来处理的话，那整个系统真的就与NAS无异了。所以MDC上的改进后的NFS服务端模块并不处理实际数据IO，而只负责应答文件的其他属性信息。既然需要客户端自行向后端的LUN发起实际IO操作，那么客户端就必须知道对应的文件段到底存放在哪个LUN的哪些Block中。这些信息只有MDC上的EXFS才知道，需要将这些信息传达给客户端，而传统NFS服务端并不具备这种功能，所以需要另外一个模块专门传达这种文件—块映射信息，这个模块就是被新加入的层次——Layout Daemon。

比如，客户端将某个SAN FS文件系统挂载到了/mnt下，某时刻客户端某应用程序发起“读取/mnt/a.txt的从Byte1024到Byte2048之间的字节”这种请求的话，那么客户端上的代理首先要向MDC上的Layout

Daemon查询以获取到/mnt/a.txt的Byte1024到Byte2048这段字节对应在后端LUN的具体Block地址（当然同时也可能会向MDC上的服务端查询一些此文件的其他属性信息）。对于追加写请求，MDC首先进行空间分配，然后将分配之后的存储空间地址信息通知给客户端，客户端向目标地址做写入动作。

Layout Daemon是MDC上负责与Client端通信的使者，它与Client端之间的交互协议一部分是类似NFS的协议，另一部分则是私有协议。

3. 客户端上的元数据信息获取者——eNFS模块

eNFS的取名意为Enhanced/Extended NFS。eNFS模块相当于客户端上的NFS Client，但是它比普通NFS Client多出一部分功能，也就是它除了需要与MDC上的改良NFS服务端通信之外，还需要与Layout Daemon交互以获取实际Block地址段信息。在获取到实际地址信息之后，eNFS便将这些信息传递给另外一个下层模块——VDMAP。

eNFS的作用方式和思想与pNFS类似，这也是其名曰eNFS的原因。

4. 客户端上的实际IO执行者——VDMAP模块

VDMAP，即Virtual Disk MAP。它的一个作用是管理和映射底层的LUN，作为一个简单的卷管理器。另外，这个模块还负责执行最终由eNFS下发的块级别IO信息。这个模块有两种形态：第一种是可以直接访问后端LUN的客户端所具有的形态；第二种则是无法直接访问LUN的客户端所具有的形态。对于无法直接访问LUN的客户端，其上的VDMAP模块可以向前者的VDMAP模块发起IO请求，由前者代为执行对相应卷的IO。

思考： 不知道读者有没有感觉到，其实文件级的集群存储系统的架构，不管是有IO节点的还是直连SAN访问的，它们本质上是殊途同归的，不管使用什么样的协议，什么样的架构，什么样的硬件，其实它们的本质思想就是MDC+并行直接访问。

另外，可以将SAN文件系统看作是一个NAS系统的改进产品。这个结论可能让人大跌眼镜，将NAS变为SAN，然后又改进？可是事实就是如此。所以说，存储领域的协议本来就是相通的，本质是殊途同归的。

15.11 集群的本质——一种自组自控轮回的Raid

纵观集群SAN与集群NAS系统，是否可以提取出一些共性的、最纯粹的东西来呢？如图15-89所示，左边为一台传统的双控制器的磁盘阵列系统精简架构图，两个控制器通过FCAL或者SAS网络共同控制着后端的多块磁盘，多块磁盘组成某种Raid类型，比如Raid 10、Raid 5等，数据被均衡打散地分布到Raid组中的所有磁盘中；而右边则是IBM XIV 集群存储系统的精简架构图，可以看到两者有什么类似了么？

图15-89 集群存储系统的本质（1）

先看看拓扑图：前者是控制器与磁盘通过某种网络比如FCAL或者SAS来连接通信，后者是前端节点与后端节点也通过某种网络连接通信。再来看IO执行过程，前者的数据IO过程是：控制器将IO下发给各个磁盘，磁盘执行IO，将结果返回给控制器，控制器再将结果返回给主机；而后者执行IO过程的过程是：前端接口节点接受主机的IO请求，将IO下发给自身磁盘或者后端数据控制器节点，自身磁盘或者后端数据控制器节点执行IO，将结果返回给前端接口控制器，接口控制器再将结果返回给主机。可以看到这两者执行IO的过程是类似的。那么再来看看数据分布的方式，XIV在所有磁盘中打散分布数据，本质上是一种Raid 10；而前者如果做成Raid 10，那么也是将数据同样打散分布。两者的区别则是：前者是控制器将IO下发给磁盘，而后者则是前端控制器节点将IO下发给后端控制器而不是直接下发给后端磁盘。

如图15-90所示为其他典型集群场景，是否可以认为所有控制器组成了一个大的Raid系统呢？当然可以！每个控制器下面的磁盘做成Raid，然后多个控制之间再做成Raid 0，或者全局磁盘做成一个灵活的

Raid 10。

图15-90 集群存储系统的本质（2）

那么是不是可以抽象成三个角色的两个层次：第一层Raid和第二层Raid、第一层网络和第二层网络、第一层控制器和第二层控制器？所谓第二层控制器是指集群功能本身这个“虚拟控制器”。

提示：目前有不少厂商的宣传用语中已经出现了“网络Raid”这个名词，其实就是指一种轮回的表现。那么我们是不是可以说，集群就是网络上的Raid，也就是Redundant Array of Independent Node，Rain？

15.11.1 三统理论

如果说客户终端访问服务器的网络为传统的以太网LAN，是第一网，也就是业务网，那么服务器访问存储系统所使用的网络就是第二网，也就是SAN，SAN可以基于以太网或者FC等网；而如图15-91所示，用于存储集群以及主机集群内部通信和数据传输的网络，已经形成了一个第三网。

图15-91 第三网

这里有个观点，叫做三统，哪三统呢？首先是集群的统一，大家知道目前有各种各样的集群，比如计算集群、存储集群，存储集群中又分为汲取SAN、集群NAS、分布式文件系统、集群文件系统等，如此多样的集群，其本质无非就是一堆x86的节点，用某种网络连接起来后面挂了大量磁盘的，就是存储集群中的节点，拥有大量CPU和内存的，就是计算节点，如果两者皆有，那就是统一集群了，如图15-92所示。

图15-92 三统理论示意图

为何计算与存储以前要分开呢？因为以前的DAS直连存储性能和容量均跟不上，而且属于孤岛形态，这限制了存储的发展，必须要将其与计算分开独立发展。所以存储后来先发展为双控制器传统网络存储，此时计算与存储无法合体；再后来，外置存储发展到集群化形态之后，虽然其表象仍然是分的，但是其里面却是合的，对外合为一体的。此时，计算与存储集群经历了长久分开之后，也必将会重新合体，寻回其本源。大家可以看到这是一个轮回和分分合合的过程。如今，存储系统正在向集群化发展，而计算也是集群化，那么计算集群与存储集群就可以完美的被融合起来了，形分神合。这种形态也属于之前提过的“自助型存储集群”。除了主机集群与存储集群的合体之外，集群SAN与集群NAS其实也可以统一，目前很多厂商都推出了块虚拟化产品，它们的LUN在后端其实就是一个文件，可以被打散存放在底层磁盘各处。既然SAN设备底层都使用类文件系统来管理了，那么SAN与NAS的后端其实就已经被统一了，剩下的，就是前端访问协议的统一了（见下文）。此外，集群硬件也将变为一个平台，其上的各种协议、应用，则变成了一种服务，比如SAN服务、NAS服务；而分布式文件系统则是集群NAS的支撑层，其本身与集群NAS属于一种本质上的东西。至此，集群硬件形态与上层软件充分解耦。

其次是访问协议的统一。既然集群已经变为一个通用集群，那么访问这个集群的方式也应该被融合。上文中曾经提到过，文件与块的本质其实是一样的，只是组织与访问方式不同罢了。如今块虚拟化的存储系统比比皆是，它们无一例外都将LUN当作一个文件一样来对待，恨不得直接在纯种文件系统中用文件虚拟出一个LUN来。既然这样，底层其实是被文件系统给统一了，那么外围的访问方式上，也应该被统一。本质

上讲，不管是块还是文件，其实它们都用同一种协议访问：操作码、目标、起始偏移、长度。对于块访问，目标就是LUN ID，而对于文件，目标就是某路径，比如/a/b/c.txt，那么是否有一种东西来屏蔽目标的不同呢？其实早就有这种协议，说到这里大家可能悟到了，这就是对象存储系统，对象存储协议就是将文件与块访问大统一的最佳候选协议了，只要时机成熟，文件、块大统一的访问方式必将席卷存储技术领域。块与文件这两种访问协议分开太久了，有合的趋势与欲望，底层技术也很给力。其实对存储协议早在20世纪80年代就被提出了，时隔30年，如今终于有了用武之地，就是利用对象协议，可以将文件与块的访问完美地融合统一起来。如果真的可以用对象存储做到统一，那么主机端会出现一种新的HBA，即OSD HBA，其将OSD Initiator集成到硬件中，存储对象既可以表现为一个目录，又可以表现为一个卷。

最后，就是网络的统一。不管第一网、第二网还是第三网，如果有一种网络可以同时满足需求，那么为何不统一呢？比如以太网。

做到这三统，这才是真正的统一存储，而不是同一个机头同时出块和文件协议。这就叫统一存储？噱头而已，看似统一，实则意义不是很大。

15.11.2 并行的不仅可以是文件

在前文介绍对象存储系统时，有一个思考，里面提到了其实文件IO与块IO的本质是一样的。既然本质相同，而分布式文件系统可以实现主机端的并行访问以提高效率，那么为何主机不可以用块协议来并行访问一个Scale-Out的SAN存储系统呢？当然可以了，没人规定不可以，只是还没有形成一个标准而已。

这方面，尖兵厂商Infortrend的产品ESVA率先打破了常规。其ESVA产品对主机侧提供了一个Load Balance Driver（LBD），也就是并行访问客户端，可以分别连接到ESVA集群中到每个节点来并行地访问数据。ESVA集群中的节点各自连接到FC交换机上。ESVA集群中有一台Master节点，统管集群中的卷元数据，其他都为成员节点，这个思想与分布式并行文件系统完全一致。当使用了LBD的时候，主机客户端会从所有节点中认到全局虚拟的LUN，并可以直接与所有节点通信读写数据；当不安装Load Balance Driver的时候，主机客户端只能通过Master节点来挂载对应的LUN，当需要访问的数据恰好落在成员节点中时，则数据必须经过Master节点的中转，会多耗费一次经过FC交换机的转发。

块级别的并行比文件级并行来得更容易，因为分布式文件系统中的文件存放位置与映射关系错综复杂，随时在变化，而块级存储中的LUN的位置相对于分布式文件系统来讲并没有那么多的变化，几乎分配完之后就恒定在对应的节点中，这样，阵列与主机端并行客户端之间需要同步更新的LUN数据布局元数据信息就很少，效率很高。

再分析下去，ESVA的并行客户端又似乎像是一个分布式卷管理系统，这里的卷其实就是LUN，只不过一个LUN会被分布到多个集群节点中存放。所以，其相当于同时提供了一台台的单独阵列，而同时又给主机上提供了一个处于主机操作系统内核卷管理更下层位置的卷管理层，将识别到的多个独立的LUN虚拟成一个Raid0的大空间。这样做存在一个风险：一旦这些独立阵列中的一台出现故障，那么整个系统的数据不再可用。但是话又说回来，如果这个系统是一个单阵列，你把所有数据都放到这台阵列中保存，那么它出问题的几率会有多少？从这种角度来讲，不管系统中有多少台阵列，与单阵列出问题的几率是相同的，而单阵列出问题导致停机的几率是非常低的。即便这样，为了增加安全系

数，厂商也实现了所谓网络Raid，即在集群中的所有节点之间再做一层Raid 5来防止单个节点故障，这样也就成了Raid for Rain。这种Raid的计算也有两种方式：一种是直接在主机端计算好之后，并行地写入每个节点，这种做法就变成了彻头彻尾的软Raid 5了，比如Windows下的动态磁盘；另一种则是由集群中的存储节点来自行计算，这样势必要浪费大量的集群内部通信网络带宽，而且性能也不会很好。

对象存储OSD协议或许有希望在一段时间之后真正成为块级并行访问的标准协议，因为它既像块，又像文件，而分布式文件系统已经有了标准访问协议，那就是pNFS，pNFS显然不适合块级，那么OSD或许就是最佳的候选对象了。

另外，有了并行访问客户端之后，之前的多路径软件也就可以被统一了，因为并行客户端自然会考虑多路径Failover和Failback的问题，之前多路径软件自身所作的负载均衡基本上都没有太大的意义，因为对于传统双控阵列来讲，LUN的工作控制器只有一个，在这个前提下，多路径负载均衡是不具太大意义的。但是在多控的Scale-Out分布式架构下，多路径软件对集群中的每个节点都有维护一条路径，那么这个时候，多路径软件已经不是传统意义的多路径了，并行访问上升为关键点，而此时，多路径软件就彻底成为一个并行访问客户端了，而不应该再叫多路径软件。在所有路径工作正常的条件下，主机可以直接从集群中的每个节点读写数据，集群内部通信网的负载是非常低的，仅用于同步一些元数据状态等，一旦主机到集群的某条或者某几条路径失效，那么集群中对应的节点就与主机失去了连接，这些节点中的数据只能通过集群内部通信网被传输到与主机尚有连接的那些节点中，通过这些节点将数据转发给主机，此时内部通信网的流量就会增加。

15.11.3 集群底层与上层解耦

上文中提到，集群硬件层可能会变为一个平台，承载各种集群软件层。就像一个卖盒饭的，为了盛饭，他不仅要自己把饭做好，还得把盒子做好，然后把饭盛到盒子里一起卖。但是他可能发现自己做饭盒实在不在行，只想专心做饭。这时，有个人专门做饭盒，他们一拍即合，相互合作。这是一种软硬分开的表现。

也就是说，比如某分布式文件系统，如果要把它包装为一款可交付的产品当然可以，只买软件光盘即可。但是用户的硬件是各种各样的，兼容性不一，软件安装其上之后可能会出现各种问题，这就是现在的软件厂商争相给自己的软件捆绑到硬件中打包出售的原因之一。这就表现为一种软硬相合。

分久必合，合久必分，又应了这句老话。软件和硬件这两者总是在分分合合中螺旋上升发展。两者也会出现相互争抢的局面，比如硬件总是想从软件层面offload下来更多的功能，比如TCP/IP Offload, iSCSI Offload, XOR Offload等，而软件似乎也在向硬件争抢，比如软Raid，软卷管理层等。

对于硬件集群平台与其上的软件集群形式（比如集群文件系统、集群SAN等），它们两者如果真的做到了解耦，那么也会发展出这种争抢态势。比如底层硬件想去offload上层软件集群中的一些机制，如消息通信机制、错误监测与恢复联动机制等。硬件提供一个标准集群，比如基于PCI-E交换网络的集群，基于Infiniband的，基于以太网的，等等，不管底层是Infiniband还是以太网，底层硬件集群会将其抽象封装为标准的接口，上层的集群软件可以更加专注于高层功能，比如数据排布方式以及效能等层面的研发与提升，底层这些通用化的机制，最终将被固化为标准，从而获得所有厂商的支持。

15.11.4 云基础架构

而如果再往上走一层，那就是云了。整个集群作为一个基础架构服务的提供者，其上覆盖资源管理层，再覆盖一层业务展现层，就可以作为一个云来提供各种各样的IT服务，如图15-93所示。

图15-93 云基础架构

世界就是一个集群。任何目前所理解的物质，都是由更小的“粒子”或者说基石组成的集群。或者说，任何“物质”，其实都是由世界本源的基本公式叠加而成的庞大算式。任何逻辑，都是这些“物质”即算式的叠加演算过程。逻辑是“物质”变化过程的体现。任何对逻辑的改变，最终会体现对“物质”的改变，而“物质”的改变，又体现为逻辑。万物皆和合。个体的无意识性活动组成了整体的有序，冥冥中总有一个奥妙之物来指引所有个体的行动，但是个体自身却感觉不到，或者正逐渐地隐约感知到！

15.12 纯软Scale-Out SAN

Infortrend的ESVA存储系统是比较特别的一款集群SAN存储系统。它的特别体现在，其Scale-Out扩展方式完全是利用主机端的一个虚拟卷Filter Driver实现的，也就是靠这个卷驱动来把所有连接到该主机上的所有节点上的资源整合起来，相当于一个定制化的LVM。但是配置依然是在存储端来配置。ESVA中的“V”就代表虚拟的意思，指的就是用这个卷驱动来将所有节点的空间整合起来。节点可以随进随出，添加新节点之后，原来的数据会被自动均衡到这个节点上，同样，踢出节点前，数据会先被迁移到其他节点上，然后再踢出。这种纯软Scale-Out SAN集群方案的好处类似于支持并行访问客户端的集群文件系统，客户端可以并行访问集群中所有节点，而传统集群SAN方案下，客户端只能从集群中的某个节点来读写数据，虽然可以使用多路径软件来变相地“均衡”，但是客户端不知道要访问的数据到底在哪个节点上，如果将一个IO发送给一个并没有对应这笔IO数据的节点，那么这个节点一样要经过内部交换网络从其他节点处取回数据再发送给客户端，这个过程耗费了无谓的资源。而ESVA纯软Scale-Out方案，客户端的卷驱动是知道每一笔IO到底要发给哪个节点的，直接有的放矢地从对应节点取数据，提高了性能。

如图15-94所示。ESVA将整个系统分为多个Pool，每个Pool可以由位于不同节点的不同磁盘组合而成，可以向任何Pool中添加任何节点的任何磁盘。

图15-94 设置一个池

如图15-95及图15-96所示，可以向Pool中添加新的节点及磁盘，也

可以从Pool中删除节点或者磁盘。如果将前者称为Scale-Out，后者则可以称为Scale-Back。

图15-95 添加或者删除节点

图15-96 添加或者删除磁盘

如图15-97所示，选择将新加入的磁盘做成何种类型的Raid组，并选择数据重新均衡时候耗费的系统资源优先级。然后系统会自动完成重新均衡。

图15-97 重新均衡

15.13 互联网运营商的特殊集群——NoSQL

数据海量集中的地方，无疑是互联网运营商（或称网络服务提供商，NSP）的数据中心了。这里代表了存储技术发展的最前沿。也正是因为海量数据存储与访问，使得传统存储架构、成本已经无法满足需求了。比如，每秒几十万次的随机IOPS，每秒10GB的流量，如果使用传统高端存储，那成本将是高不可攀的，而且很多时候需要多台存储才可以满足，而且后期的扩展性以及扩容成本都是不可接受的，NSP当然不愿意骑虎难下。正因有如此强烈的业务驱动，导致各NSP逐步采用分布式系统来构建其底层文件系统及数据库。比如先驱Google的GFS+Bigtable，以及后续追随者Amazon的Dynamo、Facebook的Cassandra、Microsoft的Azure、Yahoo的PNUTS、淘宝的TFS+Tair等。这些系统都是动辄数百甚至数千个节点组成的分布式集群。

1. 弱环境

这些系统无疑都是使用廉价定制化的PC来搭建的，单个节点各自有各自的计算资源和存储资源，互不共享，也就是Share-Nothing。而且单节点的可靠性不高。所以NSP分布式系统的硬件环境为一种“弱环境”，为了保证最终的可靠性，那么就必须在软件层面实现强环境，比如将数据块保存3个副本在三台独立的机器上。为何要是3而不是2呢？因为如果只有两份副本的话，发生数据丢失的可能性依然很大，比如一旦某个节点故障或者某个磁盘故障，此时对应的数据就处于危险边缘。对于一个1000节点的集群，每周大概有20块硬盘损坏，保存同一个数据的2份副本的两个硬盘同时失效的可能性也是有的。所以如果保留3份副本的话，危险程度会大大降低。

2. CAP理论

传统的关系型数据库已经无法在保证性能的前提下达到如此高的扩展性，因为传统关系型数据库是要保证强一致性的（比如用户A写入的数据，用户B立即就能看到等等）。所以，一致性、性能、扩展性，这三者是不能够同时兼顾的，只能任选其中两者（这便是著名的CAP理论）。而NSP的某些对外业务所需要的就是超高性能和扩展性，那么其只好适当牺牲一致性了，因为就大部分互联网服务而言，数据短暂的不一致是可以接受的，比如DNS解析，更换主机之后可能并不能实时刷新到全网范围。同样，由于数据块保留3个副本，那么是当所有副本都同步写入对应节点之后再向客户端返回成功信息，还是只写一份就返回成功信息呢？后者肯定是有利于性能，但是却牺牲了一致性，比如当某份副本还没有更新到对应节点时，有另一个客户端从这个尚未被同步的节点发起对这份数据的读，那么这个客户端将读到过期的数据，这就发生了数据不一致。CAP理论是指Consistency、Availability以及Partition Tolerance，分别指数据访问时的一致性、可用性（请求总是可以被执行和返回，不超时。可视为性能）以及分区容忍性。前两者大家都理解，最后一个“分区容忍性”是指大规模集群一旦遇到网络或者其他类型故障导致整个集群被裂为多个孤岛，此时集群是否依然正常提供服务，这一点可以视为扩展性中的一个子需求。

3. NWR模型

有什么办法可以保证一致性呢？办法当然有，有一个著名的模型叫做NWR模型，可以在一致性与性能之间实现巧妙的均衡。N表示对应数据块要保存的副本数，W表示成功了写入几份就可以向客户端返回成功信息，R表示为了保证读时候的一致性，客户端必须从保存这份副本的

N个节点中的几个同时读出数据。比如 $N=3$ ， $W=1$ ，也就是刚才那个场景，那么此时如果 $R=1$ ，则不能保证一致性，因为此时R可能从另外两个保存这份副本的节点读取，但是也有可能从当初W的那个节点读取，那么就是一致的，但是不能保证每次都从当初W的节点读取；而如果 $R=2$ ，同样不能保证；但是R如果 $=3$ ，那么就可以保证，读出的这3份数据，至少有一份是最新的，那么通过判断数据块的timestamp时间戳就可以判断这3份中哪一份（或几份）是最新的，从而丢弃剩余的不一致的副本。而如果 $W=2$ ，也就是必须同步的写成功2个副本才返回成功信号，那么读的时候就可以读出2份就能保证至少有一份是最新的了。所以，只要 $W+R>N$ ，即可保证读一致性。有人问说，如果读的时候只从当初写入的那个节点读不就都一致了么？是的，但是这样做就无法保证性能了，所有节点均可同时写入同时读取同一份副本，这样性能是最高的，只不过牺牲了一致性；如果要保证一致性，那么就采取NWR模型来调和到底是要读性能高（增加W的值）还是要写性能高（降低W的值），读写性能都高的话就得完全牺牲一致性，这就是NWR模型的魅力所在。

4. 水平切分与垂直切分

数据在集群中的分布有两种方式，第一种是水平切分方式，比如一张表，可以水平被裂开为多张表，每张表都保存有原来的列，每张表被分别存储到集群中的一个节点中（并保留其他两个副本）；而垂直切分相当于把一张表竖直裂开为多张表，原来的多个列被拆开，分别存储在集群中的一个节点中（并保留其他两个副本）。这样做相当于把数据打散，从而获得超高的随机查询性能。垂直切分相当于是基于业务层面的切分，具有一定的人为介入度；而水平切分则相当于全局层面的数据盲切分。

5. NoSQL

此外，这种弱一致性的数据库集群一般都不支持事务。同时，也不支持传统关系型数据库的关联查询动作，也就是说，由于完全分布式，所以多个节点之间的数据如果想要进行关联查询是很不经济的，会导致整个系统慢的不可想象，这就是上文中曾经提到的，MPP有MPP的适用场合，如果每笔查询或者IO（比如MPP架构商用存储系统中的大块连续IO）都需要所有节点提供数据分片，那么此时并发性能将会非常差。所以，这类去关联性的、弱一致性的轻量级的分布式数据库系统，就属于NoSQL系统，即“抛弃传统SQL”或者“Not Only SQL”的意思。

第16章 未雨绸缪——数据保护和备份技术

- 数据保护

- 快照

- CDP

- 数据备份

- 备份通路

数据是表示信息的信息。数据丢失或者损坏，信息也就丢失或者被扭曲了。为了防止数据意外丢失或者损毁，人们想出了一系列的方法来保护当前的数据。此外，为了最大程度地保护数据，人们还在其他地方保存数据的一份或者多份副本。

数据保护和数据备份看似简单，实则很复杂。数据保护和备份已经成为存储领域的一门分支学科。本章就这个分支作简要介绍，并给出几个案例。

16.1 数据保护

数据保护，就是对当前位置上的数据进行备份，以防突如其来的磁盘损坏，或者其他各种原因导致的数据不可被访问，或者部分数据损坏，影响到业务层。备份后的数据，可以在数据损毁之后恢复到生产磁盘上，从而最大程度地降低损失。

从底层来分，数据保护备份可以分为文件级的保护和块级的保护。

16.1.1 文件级备份

文件级的备份，即备份软件只能感知到文件这一层，将磁盘上所有的文件，通过调用文件系统接口备份到另一个介质上。所以文件级备份软件，要么依靠操作系统提供的API来备份文件，要么本身具有文件系统的功能，可以识别文件系统元数据。

文件级备份软件的基本机制，就是将数据以文件的形式读出，然后再将读出的文件存储在另外一个介质上。这些文件在原来的介质上，存放可以是不连续的，各个不连续的块之间的链关系由文件系统来管理。而如果备份软件将这些文件备份到新的空白介质上，那么这些文件很大程度上是连续存放的，不管是备份到磁带还是磁盘上。

磁带不是块设备，由于机械的限制，在记录数据时，是流式连续的。磁带上的数据也需要组织，相对于磁盘文件系统，也有磁带文件系统，准确来说应该叫做磁带数据管理系统。因为对于磁带来说，它所记录的数据都是流式的、连续的。每个文件被看作一个流，流与流之间用一些特殊的数据间隔来分割，从而可以区分一个个的“文件”，其实就是一段段的二进制数据流。磁带备份文件的时候，会将磁盘上每个文件的

属性信息和实体文件数据一同备份下来，但是不会备份磁盘文件系统的描述信息，比如一个文件所占用的磁盘簇号链表等。因为利用磁带恢复数据的时候，软件会重构磁盘文件系统，并从磁带读出数据，向磁盘写入数据。

在2000年之前，很多人都用磁带随身听来欣赏音乐。而2005年之后，就很少看到带随身听的人了，大都换成了MP3或MP4或者手机等设备。随身听用的是模拟磁带，也就是说它记录的是模拟信号。电流强，磁化的就强；电流弱，磁化的就弱。磁转成电的时候也一样，用这种磁信号强弱信息来表达声音振动的强弱信息，从而形成音乐。MP3则是利用数字信息来编码声音振动强弱和频率信息。虽然由模拟转向数字，需要数字采样转换，音乐的质量相对模拟信号来的差，算法也复杂，但是它具有极大的抗干扰能力，而且可以无缝地与计算机结合，形成能发声的计算机（多媒体计算机）。

录音带、录像带，都是模拟信号磁带。用于文件备份的磁带，当然是数字磁带，它记录的是磁性的极性，而不是被磁化的强弱，比如用N极来代表1，用S极来代表0。

如果备份软件将文件备份到磁盘介质或者任何其他的块介质上，那么这些文件就可以是不连续的。块设备可以跳跃式地记录数据，而一个完整数据链信息，由管理这种介质的文件系统来记录。磁盘读写速度比磁带要高得多。

近年来出现了VTL，即Virtual Tape Library（虚拟磁带库），用磁盘来模拟磁带。这个概念看似复杂，其实实现起来无非就是一个协议转换器，将磁盘逻辑与磁带逻辑相互映射融合，欺骗上位程序让其认为底层物理介质是磁带，然后再按照磁盘的记录方式读写数据，这就是虚拟

化的表现。这种方法，提高了备份速度和灵活性，用处很大。

16.1.2 块级备份

所谓块级的备份，就是备份块设备上的每个块，不管这个块上有没有数据，或是这个块上的数据属于哪个文件。块级别的备份，不考虑也不用考虑文件系统层次的逻辑，原块设备有多少容量，就备份多少容量。在这里“块”的概念，对于磁盘来说就是扇区Sector。块级的备份，是最底层的备份，它抛开了文件系统，直接对磁盘扇区进行读取，并将读取到的扇区写入新的磁盘对应的扇区。

这种方式的一个典型实例，就是磁盘镜像。而磁盘镜像最简单的实现方式就是RAID 1。RAID 1系统将对一块（或多块）磁盘的写入，完全复制到另一块（或多块）磁盘，两块磁盘内容完全相同。有些数据恢复公司的一些专用设备“磁盘复制机”也是直接读取磁盘扇区，然后复制到新的磁盘。

基于块的备份软件，不经过操作系统的文件系统接口，而通过磁盘控制器驱动接口，直接读取磁盘，所以相对文件级的备份来说，速度加快很多。但是基于块的备份软件备份的数量相对文件级备份要多，会备份许多僵尸扇区，而且备份之后，原来不连续的文件，备份之后还是不连续，有很多碎片。文件级的备份，会将原来不连续存放的文件，备份成连续存放的文件，恢复的时候，也会在原来的磁盘上连续写入，所以很少造成碎片。有很多系统管理员，都会定时将系统备份并重新导入一次，就是为了剔除磁盘碎片，其实这么做的效果和磁盘碎片整理程序效果一样，但是速度却比后者快得多。

16.2 高级数据保护方法

16.2.1 远程文件复制

远程文件复制方案，是把需要备份的文件，通过网络传输到异地容灾站点。典型的代表是rsync异步远程文件同步软件。这是一个运行在Linux下的文件远程同步软件。它可以监视文件系统的动作，将文件的变化通过网络同步到异地的站点。它可以只复制一个文件中变化过的内容，而不必整个文件都复制，这在同步大文件的时候非常管用。

其实FTP工具也是一个很好的远程文件复制工具，只不过不能做到更加灵活和强大而已。

16.2.2 远程磁盘（卷）镜像

这是基于块的远程备份，即通过网络将备份的块数据传输到异地站点。远程镜像（远程实时复制）又可以分为同步复制和异步复制。同步复制，即主站点接受的上层IO写入数据，必须等这份数据成功地复制传输到异地站点，并写入成功之后，才通报上层IO成功消息。异步复制，就是上层IO主站点写入成功，即向上层通报成功，然后在后台将数据通过网络传输到异地。前者能保证两地数据的一致性，但是对上层响应较慢；而后者不能实时保证两地数据的一致性，但是对上层响应很快。

所有基于块的备份措施，一般都是在底层设备上进行，而不耗费主机资源。

现在几乎各个盘阵厂家的中高端产品，都提供远程镜像服务，比如IBM的PPRC、EMC的SRDF、HDS的Truecopy、Netapp的SnapMirror

等。

16.2.3 快（块）照数据保护

远程镜像，或者本地镜像，确实是对生产卷数据的一种很好的保护，一旦生产卷故障，可以立即切换到镜像卷。但是这个镜像卷，一定要保持一直在线状态，主卷有写IO操作，那么镜像卷也有写IO操作。如果某时刻想对整个镜像卷进行备份，需要停止读写主卷的应用，使应用不再对卷产生IO操作，然后将两个卷的镜像关系分离，这就是拆分镜像。切分过程是很快，所以短暂的IO暂不会对应用产生太大的影像。

拆分之后，可以恢复上层的IO。由于拆分之后已经切离的镜像关系，所以镜像卷不会有IO操作。此时的镜像卷，就是主机停止IO那一刻的原卷数据的完整镜像，此时可以用备份软件将镜像卷上的数据备份到其他介质。

拆分镜像，是为了让镜像卷保持拆分一瞬间的状态，而不再继续被写入数据。而拆分之后，主卷所做的所有写IO动作，会以bitmap的方式记录下来。bitmap就是一份位图文件，文件中每个位都表示卷上的一个块（扇区，或者由多个扇区组成的逻辑块），如果这个块在拆分镜像之后，被写入了数据，则程序就将bitmap文件中对应的位从0变成1。待备份完成之后，可以将镜像关系恢复，此时主卷和镜像卷上的数据是不一致的，需要重新做同步。程序会搜索bitmap中所有为1的位，对应到卷上的块，然后将这些块上的数据同步到镜像卷，从而恢复实时镜像关系。

可以看到，以上的过程是十分复杂烦琐的，而且需要占用一块和主卷相同容量大小的卷作为镜像卷。最为关键的是，这种备份方式需要停

掉主机IO，这对应用会产生影响。而“快照技术”解决了这个难题。快照的基本思想是，抓取某一时间点磁盘（卷）上的所有数据，而且完成速度非常快，就像照相机快门一样。

下面介绍快照的底层原理。

1. 基于文件系统的快照

自然界是不断变化的。可以用照相机抓拍到某个时间点的瞬间影像。为什么要对自然界照相？为了留下美好的记忆。计算机存储的电子数据也是在不断变化的，为了抓拍下某一时间点某份数据集的内容，需要一种“数据照相机”。为什么要对电子数据进行抓拍？为了预防数据被“污染”，或者逻辑上被损毁，比如病毒大范围感染、配置信息崩溃、系统崩溃等。拍下一些历史时刻的数据内容留底。自然界照片至少现在还不可以回滚，但是电子数据的照片确是可以回滚的，并且可以从这个时间点为起点重头来过，或者从这个时间点的数据影像创建出多份平行的存储空间（克隆）。

A兄提出的办法：“快速将某文件系统内的文件全部复制到另外一个地方存放，这份复制出来的文件集就是源文件系统在复制开始那一时刻的快照！”这种做法有什么问题？如果文件容量很大，复制需要很长时间，而这段时间内源文件系统内有数据发生更改怎么办？抓拍移动的景物，曝光时间越长，得到的照片就会重影、模糊甚至花脸！对于电子数据也一样，如果曝光（复制）时间过长，曝光（复制）的过程中被拍摄的对象有移动（数据被更改），那么所得到的照片（数据快照）就是一份花脸照片（得到的数据集前后不一致）。

B兄提出了改进方法：在复制时禁止一切针对源文件系统的更改操

作，即禁止一切写IO。等复制完成时再恢复。看来B兄是个不合格的摄影师。自己摄影技术太差，倒霉的是来拍照的人，强行让客户静止不动长达几分钟，霸王条款！

C兄出马了：“从本质上解决这个问题，必须降低曝光（数据复制）时间！同时只需被拍照者（应用程序）只静止一瞬间就可以了！”

D兄出面了：“C兄说得很对！我有个办法可以一举两得地保证这两个苛刻需求的落地！”D兄有何秘籍？万变不离其宗！其“宗”为文件系统元数据链。牵一发而动全身，只要抓住了文件系统元数据链，就等于掌握了目前这个文件系统内的所有信息。而对于一个文件系统来讲，其元数据量相对整体数据量是很少的。如果在复制的时候只复制元数据链，也就是整个文件系统的总图纸，那么就变相地拥有了一份在复制一刹那这个文件系统的所有内容的组织结构树。

顿时，A、B、C三位老兄开始狂轰滥炸：“只复制元数据链，那么复制过程是可以大大降低了，但是如何解决在复制完成之后仍然允许用户程序更改源数据呢？一旦有数据被更改，那么复制出来的图纸就相当于作废了，因为图纸与实际内容已经对不上号了！”

D兄从容应对：“我自有办法。源文件数据当然不能让它就这么被覆盖掉。有以下两种办法来保证。”

办法一：元数据复制完成之后所有针对源文件系统的更改文件块均将其重定向到一块空余的空间内存放，并且在源文件系统的元数据更新对应的指针条目来告诉系统说源文件系统中某文件的某些块的最新内容其实被重定向存储到了新的空间内的某某地址。这样就可以永久地将快照创建瞬间的数据全部冻住，之后所有更改都被重定向到剩余空间存

放。此时系统内保存着两套元数据链，一套是源文件系统的，会被不断地更新；另一套则是快照时保存下来的，永不更新，而且其指向的实际存储位置上的数据也不会被覆盖。随着源数据被不断覆盖而产生的重定向写动作，空闲空间内会逐渐产生一个当前时间点文件系统全部数据的影像，而拍照那一刻的历史时间点的源文件系统影像则永久被冻住为快照生成瞬间的历史时间点影像。这种思路叫：Redirect on First Write（RoFW）。之所以只在首次覆盖写时重定向，是因为只要重定向出去之后，源数据块后续的覆盖就会直接在重定向之后的块地址上直接覆盖之前被重定向出来的块了。

办法二：元数据复制完成之后，所有针对源文件系统中文件的覆盖写操作均照常执行，但是在覆盖对应的数据块之前，需要将被覆盖的数据块内容复制出来，存放到一个额外的空闲空间内存放，并且在之前被复制出来的元数据链中更新这个新指向记录，将历史的一部分指向被复制出来的数据块，历史的另一部分则与现在是重叠存在的。系统中依然有两套元数据链，一套指向当前，另一套指向历史。这样处理之后，源文件系统存储空间中永远都是最新的数据，而历史数据会随着覆盖写入操作的不断进行而被逐渐搬出原存储空间，存入空闲空间，最后在空闲空间内生成为一个快照生成那个瞬间的历史影像。这种思路叫：Copy on First Write（CoFW）。之所以只在首次覆盖写时复制，是因为只要首次复制出去之后，源数据块后续的覆盖写动作就会直接在源数据集中对应地址上直接覆盖了，因为历史版本已经被保留了。

我们知道，文件系统管理思想的精髓就在于它的链表、B树和位图等结构，也就是元数据（Metadata），以及对这些元数据的管理方式。文件系统其实对底层磁盘是有点“恐惧”的，我们可以计算一下，一个100GB的磁盘上，有超过2亿个扇区。文件系统是如何管理这2亿个扇

区，又如何知道某个扇区正在使用呢？如果使用的话，是分配给哪个文件或者文件的一部分呢？

文件系统首先将扇区组合成更大的逻辑块来降低管理规模。NTFS最大每个块可以到4KB，也就是8个扇区一组形成一个簇（块，Block）。这样，2亿的管理规模便会除8，缩小到1.2千万的管理规模，虽然存储空间可能有所浪费，但是切实降低了管理成本。

其次，文件系统会创建所管理存储空间上所有簇（块）的位图文件，这个文件有固定的入口，文件系统能在1.2千万个块中快速定位到这个文件入口并读写。位图文件中每个位代表卷上的一个簇（或者物理扇区，视设计不同而决定），如果簇正在被某个文件使用，这个簇在位图中对应的位的值就为1；否则为0。

再次，文件系统还保存一份文件和其所对应簇号的映射链，这个映射链本身以及簇位图本身也是文件，也要有自己的映射链，所以针对这些重要的元数据，必须有一个固定的入口，用来让文件系统程序读入并且遍历所有文件系统元数据。通常将这个初始固定地址入口称为root inode，不同操作系统具体实现方式不同。

当向卷中写入一个新文件，文件系统首先会查找簇位图，找到位值为0所对应的簇号，并计算所需的空间，然后分配这些簇号给这个文件。它首先将文件实体数据写入对应的簇，然后再去文件—簇号映射图中更新，将新文件与其对应的簇映射关系记录下来，最后到簇位图中将这些簇对应的位的值从0改为1。如果要删除这个文件，则直接在inode链中将这个文件的inode抹掉即可，然后在簇位图中将对应簇的位值从1改为0。

文件系统并不会抹掉这个被删除文件所对应卷上的簇中的实际数据，如果用扇区读写软件来提取这个簇，就会得到这个文件的部分内容。虽然这些簇中依然有内容，但是对于文件系统来说，这些簇是可重用的，一旦有新文件写入，新文件的数据便会覆盖原来簇中的数据。

所以，对于一个文件系统来说，最重要的不是卷簇中的数据，而是文件—簇号映射链和位图等这些元数据。比如，想要破坏某个文件系统中的一个文件，我们不必费劲地修改某个文件对应簇中的实际内容，只需修改一下文件—簇号映射链中关于这个文件所对应的实际簇号的记录即可，让它指向其他簇号。

这样，文件读出来的内容就不是原有内容。此外，如果修改了文件系统中对应簇中的数据，文件系统也根本感知不到这些动作，因为它所查询的是文件—簇号映射链，它只知道某个文件对应着哪些簇，而不关心这些簇是否被改过，它想关心也无法关心。

正因为文件系统只是根据它所记录的这些映射图表来管理文件，所以才使得快照成为可能。为什么呢？

思考：如果我们想抓取某个卷在某一时刻下的全部数据，就可以像照相机一样，对一个正在移动的物体实施拍照，某一时刻，快门打开，曝光后关闭。照相机可以保存在底片上，但照下的数据如何保存呢？

如果一个卷上有100GB容量的文件，照下来之后，用另一块磁盘将数据全部复制出来，至少需要20分钟的时间，且在这段时间内，不允许再有任何数据被写入这个卷，因为我们要抓取的是这个卷在这个时刻瞬间的数据。

对于拍照移动中的物体，要想保证照片清晰没有重影，就要降低曝光时间，但是曝光时间越短，照片感光度就会越低，也会影响质量。同样，对于一个正在进行频繁IO写入的数据卷，要照下它某时刻的样子，也需要减少“曝光”时间。但是刚才已经说过，100GB的数据，复制出来需要20分钟时间，这个“曝光”时间太长了。要么物体在拍摄的时候保持静止（复制卷的时候停止IO），要么就减少曝光时间（加快复制速度），这两样，我们一样也保证不了，因为停止IO会直接影响上层应用，而且也无法在几秒钟（应用停止IO可接受的时间内）之内将100GB的数据复制出来。难道就真的没有办法了么？非也。

思考： 如果我们只花几秒甚至1秒钟的时间，把某时刻的文件系统中的映射图表，以及下级链表等元数据复制出来并保存，那么是不是就可以说，我们照下了这个时间点处卷上的所有数据呢？

绝对是的。因为文件系统只根据这些映射图来对应卷上的实际数据，所以只要有了映射图，就可以按图索骥，找到并拥有实际的数据。但是，将此时刻的映射图复制出来，我们也只得到了一个图纸而已，因为实际数据在卷上，是实实在在的物理卷，而不是在图纸上，所以我们必须保证卷上的数据不被IO写入，而同时又不能影响应用，既然不能影响应用，就要让IO继续执行，这简直是个大大的矛盾啊！

思考： 源块不能被写入新数据，这个绝对要保证，否则这张“照片”就花脸了，而同时应用的IO也必须持续不断地执行（写入），既然源块不让写入了，那能不能写入到其他空闲的地方呢？

当然可以！每当遇到需要向原卷写入新文件或者更新旧文件，文件

系统便将这些更新数据写入一个新的空闲的地方去，然后在它的映射图中加入对应的条目。由于我们已经保存了原来的映射图，而且也拥有一份不让写入的原卷实体数据，所以不怕它修改。文件系统当前的映射图（元数据）始终描述的是当前的映射关系。这样，应用继续执行IO，同时，我们也可以将照下来的原卷数据复制出去，从而得到一份某时刻的瞬时的卷数据。我们完全可以选择不复制这份快照，就让它留在那，因为快照根本不占用额外的空间，除非针对这个卷有新的IO写入，则新簇会写向其他地方，从而占用相应大小的空间，一旦原卷所有簇都被更新了，那么也就意味着在空闲空间内逐渐生成了一个完整的新卷了，其占用与原卷相同大小的空间。

图16-1所示为CoFW模式下的几个典型流程。

图16-1 CoFW快照示意图

（1）源文件系统（当前活动文件系统，即正在被应用程序读写的生产文件系统）初始状态：数据不断地被读取或者写入，实际数据块（文件块，如A、B、C、D）以及元数据链也在不断变化。

（2）之后某时刻，系统被触发了一份快照。系统将所有写IO暂挂，然后立即开始将整个文件系统的元数据链复制一份存放，复制完成后立即解除暂挂。被复制下来的这份元数据链是与当前活动元数据链完全相同的，所以此时此刻其指向底层数据块与当前活动元数据链也是相同的。

（3）之后某时刻应用程序要更改当前活动文件系统上的C数据块为C'，属于首次覆盖写，根据CoFW规则，需要先将原来的C数据块复制出来，然后再写入C'数据块。之后，在快照元数据链中将原本指向C块地

址（现在的C'块的地址）的指针改为指向被复制出来的C块的新地址上。

（4）同理，之后某时刻A被覆盖更改为A'，那么系统同样也将A复制出来并且修改快照元数据链中的指针。

（5）在这一步中，B被更改为B'，B同样被拷出。同时，应用程序对当前活动文件系统做了一个追加写动作，比如扩大了某文件之类，产生了一个新分配的文件数据块E，由于这个文件系统在快照那一刻的历史版本中并没有这个数据，所以系统并不会作任何后台复制动作。对于追加写的数据块，快照系统不做任何处理。应注意，追加写是文件系统中经常发生的事情，追加写会伴随着元数据的数量和容量的变化而不仅是指针指向的改变，也就相当于图纸扩大幅面了。但是对于LUN或者卷来讲，并没有追加写这一说，卷的容量和位置是固定的，除非某些场景下将某个LUN扩容，而这个动作也不需要后台的快照处理进程做什么事情，因为历史时刻这个LUN并没有扩容，所以当前活动LUN怎么折腾都行，只要不发生需要首次覆盖源数据块的情况，快照无须做任何处理，拿好手中的图纸，监控好首次覆盖动作即可。另外，这一步中还有一个动作，即C被覆盖写为C''，由于C块之前已经被覆盖过一次，所以这次系统直接用C''覆盖之前的C'，无须任何处理过程。

（6）当源文件系统内所有数据块都被覆盖写了一遍之后，系统内将有两套完整的文件系统数据集：一套是当前活动文件系统，另一套则是这个文件系统所对应的历史快照时刻的数据版本。

图16-2所示为RoFW模式下的系统快照处理流程。

图16-2 RoFW快照示意图

(1) 源文件系统（当前活动文件系统，即正在被应用程序读写的生产文件系统）初始状态：数据不断地被读取或者写入，实际数据块（文件块，A、B、C、D）以及元数据链也在不断变化。

(2) 之后某时刻，系统被触发了一份快照。系统将所有写IO暂挂，然后立即开始将整个文件系统的元数据链复制一份存放，复制完成后立即解除暂挂。被复制下来的这份元数据链是与当前活动元数据链完全相同的，所以此时此刻其指向底层数据块与当前活动元数据链也是相同的。

(3) 之后某时刻应用程序要更改当前活动文件系统中的C数据块为C'，属于首次覆盖写，根据RoFW规则，直接将C'数据块重定向写入到某剩余空间内。之后，将当前活动文件系统元数据链中原本指向C块地址（现在依然是C块的地址）的指针改为指向被重定向写出去的C'块的新地址上。

(4) 同理，之后某时刻A被覆盖更改为A'，那么系统同样也将A'重定向写出，并且修改当前活动文件系统元数据链中的指针。

(5) 在这一步中，B被更改为B'，B'同样被重定向。同时，应用程序对当前活动文件系统做了一个追加写动作，比如扩大了某文件之类，产生了一个新分配的文件数据块E，由于这个文件系统在快照那一刻的历史版本中并没有这个数据，所以系统并不会作任何后台重定向动作。对于追加写的数据块，快照系统不作任何处理。另外，这一步中还有一个动作，即C被覆盖写为C''，由于C块之前已经被重定向过一次，所以这次系统直接用C''覆盖之前的C'，无须任何处理过程。

(6) 当源文件系统内所有数据块都被覆盖写了一遍之后，系统内

将有两套完整的文件系统数据集：一套是当前活动文件系统，另一套则是这个文件系统所对应的历史快照时刻的数据版本。

图16-3所示为在RoFW模式下生成两个快照之后系统的处理流程示意图。

图16-3 RoFW模式生成两份快照后处理流程示意图

(1) 某时刻的状态如图中1号框所示，已经生成了一份快照，并且重定向写出了A'和C'这两个数据块。

(2) 之后某时刻，当前活动文件系统的B被更新为B'，则系统直接将B'重定向写出去。然后，系统又被触发了一份快照。系统将所有写IO暂挂，然后立即开始将整个当前活动文件系统的元数据链复制一份存放，复制完成后立即解除暂挂。被复制下来的这份元数据链是与当前活动元数据链完全相同的，所以此时此刻其指向底层数据块与当前活动元数据链也是相同的。可以看到数据块D是被三套元数据链共享指向的。

(3) 之后某时刻应用程序要更改当前活动文件系统中的A'数据块为A''，此时是否需要做一些动作？我们来看一下，对于快照1来讲，A'已经是被重定向复制出去的块了，那么对于快照1来讲是不需要重定向写，但是对于快照2呢？就不同了，对于快照2来讲，用A''覆盖A'就属于首次覆盖了，所以需要将A''重定向写出去。同时可以看到A'对于当前活动文件系统和快照1来讲已经没有用了（不指向它了），但是对于快照2来讲却是必须的，A'是快照2时刻这个块所对应的历史版本。

(4) 同理，之后某时刻D被覆盖更改为D'，那么系统同样也将D'重定向写出，并且修改当前活动文件系统元数据链中的指针。D这个数据块依然保留，因为快照1与快照2共同指向它。

(5) 在这一步中，B'被更改为B"，C'被更改为C"，B"与C"同样被重定向写出去并更改当前活动文件系统元数据链指向。可以看到D数据块仍然被两个快照共享。而当前活动文件系统的实际数据块已经全部被重定向写出去了。

还没结束。到这一步看似很完美了，给文件系统拍照的全流程已经梳理完毕，但是要考虑精益求精。在面对一个海量庞大文件系统的时候，其元数据量可能会达到上GB甚至上百GB，这都是有可能的。而此时如果对这个文件系统做快照，首次的元数据复制所耗费的时间将是不可接受的。有没有办法再降低复制量？

我们还是用万变不离其宗、牵一发动全身这两个思想来作为解决这个问题的入口。整个文件系统的“宗”就是其元数据链，那么元数据链本身有没有一个“宗”呢？对了，确实有。这个“宗”就是其根入口块，或者称Super Block。这个块的地址在底层空间上是绝对恒定的，这个块内存放有指向下一级元数据链块的指针，操作系统每次载入元数据都要从这个地址读入Super Block，从而根据其中的指针一层层地向下遍历。那么我们如果在每次快照的时候只将Super Block复制保存，是否可以呢？

当然可以。如果只复制Super Block，那么其下级的所有元数据链自身的变更，也要进入CoFW或者RoFW流程了。图16-4所示为只复制根节点Super Block并且使用RoFW模式下的文件系统快照流程示意图。

图16-4 只复制Super Block的RoFW模式示意图

(1) 源文件系统（当前活动文件系统，即正在被应用程序读写的生产文件系统）初始状态：数据不断地被读取或者写入。Super Block指向了两个一级间接inode（i1与i2）。i1与i2又分别指向了两个实际数据

块A、B与C、D。当然，实际上不可能只有这么少的间接块以及这么浅的数据链级数，这里只是一个示意图。此时此刻，A、B、C、D四个块的内容不断变化，但是i1与i2间接块的内容却不变，因为i1块的内容是A与B块所在的地址，同样，i2块中的内容其实是C与D块所在的地址，也就是说i1与i2块其实是个指针块，ABCD的内容不断在变，但是它们在底层存储空间上的地址偏移是不变的，所以i1与i2块内容也不会变。当i1或者i2间接块对应的文件发生长度变化时，i1或者i2可能会变化，比如新指向一个E块，i1或者i2中就需要新添加一个地址指针来指向E块。

（2）某时刻，系统触发了一份快照，此时只需要将Super Block复制保存即可。

（3）随后，数据块A发生内容变更，由于RoFW模式的作用，新数据块A'需要重定向到一个新位置上，并且修改指针指向，也就是修改i1块中的指针将其指向A'块的位置，那么也就意味着，i1块的内容也发生了改变，变为i1'，那么同样也需要将i1'重定向写入新位置，牵一发而动全身，i1的指针变了，Super Block中针对原本i1的指针现在也需要指向i1'的新地址。i1'中针对B数据块的指针不变，仍然指向原B块地址。

（4）随后，数据块B发生了变更，变为B'，B'当然要被CoFW，B的上一级间接块为i1的指针也要改，由于之前i1'已经被做了RoW，所以本次只要在i1'中将B的指针指向B'新地址即可。这一步之后，i1块及其下挂的数据块AB只被快照所使用，而i2及其下挂的数据块CD仍为活动文件系统以及快照所共享。

（5）随后，C发生变更，执行与之前相同的动作，这里不再叙述。

(6) 当A、B、C、D都发生变更之后，其对应的上一级间接块也随之都发生了变更，也都被CoFW了出去，最后系统形成了两份独立的文件系统元数据链及其指向的实际数据块。

首次只复制Super Block的做法加快了复制速度，使得快照真的可以被瞬间执行，但是后续依然需要将完整的元数据进行CoFW或者RoFW，相比于首次复制全部元数据链的模式，其实需要复制的数据量长期来看是一样的，但是前者却可以更加迅速地完成快照拍摄过程。

提示： NetApp公司的WAFL文件系统利用的快照方式是很有特色的。每次快照，它只将根inode复制并保存，而不保存下级链表inode。之所以敢这么做的原因，是因为WAFL从来不会覆盖写入某个文件对应的旧块。不论是元数据还是实体数据，WAFL统统写入到卷的空闲块上（根节点inode映射图位置恒定，每次更新会覆盖写）。这样，在只复制了根节点inode之后，由于下级链表inode均不会被覆写，所以同样可以保存瞬间的snapshot。其他的快照实现方式，一般都把所有inode复制并保存，因为它们的inode都是恒定位置的，只能全部覆写。WAFL的这种模式相当于是只复制根入口的RoW（注意不是RoFW）模式的快照。

2. 基于物理卷的快照

对于基于物理卷的快照，其比文件系统快照实现起来要简单。因为LUN或者卷一般在底层磁盘上的物理位置是恒定的，而不像文件系统那样可以随意细粒度地分布。正因如此，LUN的映射元数据链并不像文件系统那样复杂，可以认为LUN的元数据就是其在底层磁盘上的起始和结束地址。

这样，在拍照时，需要复制的元数据链就更小，是真正的瞬间完成。但是完成之后就难受了，需要按照一定的粒度来做CoFW或者RoFW，而且还需要记录更改的数据映射指针，细粒度的元数据指针是文件系统的特长，现在就需要将它搬到LUN卷这一层来实现了。对于某些实现了块级虚拟化的系统如NetApp、XIV、3PAR等，它们的LUN在底层的位置都不是固定的，此时，LUN相当于一个文件，有一串类文件系统的元数据链来维护这些映射关系。所以，这些系统实现快照的原理与文件系统级快照类似。

基于物理卷的快照，相当于给物理卷增加了一个“卷扇区映射管理系统”。我们知道，卷扇区应当是由文件系统来组织和管理的，但是为了减轻文件系统负担，人们在底层卷这个层次实现快照。卷扇区都是用LBA来编号的，实现快照的时候，程序首先保留一张初始LBA表，每当有新的写入请求的时候，程序将这些请求的数据写入另一个地方（一般是一个新卷，专为快照保留的），并在初始LBA表中做好记录，比如：

原始LBA：卷A的10000号，映射到LBA：卷B的100号

以上映射条目的产生，是由于有IO请求写入数据到卷A的10000号LBA，由于做了快照，卷A在这个快照被删除之前不允许写入，所以将这个写入请求的数据，重定向写到卷B的100号LBA扇区上。值得说明的是，文件系统不会感知到这个重定向动作，FS在它的映射图中依然记录了卷A的10000号LBA地址而根本不知道还有个卷B。

此时，如果文件系统生成了一个IO请求读取，或者写入卷A的10000号LBA扇区，那么运行在卷层的快照程序便会查找快照重定向映射表，发现卷A的10000号LBA其实已经被重定向到了卷B的100号，然后读取或者写入卷B的100号扇区。由于每次IO，程序均会查找这份快

照映射表，所以增加了处理时间，降低了一些性能。这种方式称为 **Redirect on First Write (RoFW)**，意思是重定向写，也就是将更新的数据写入另一个地方，原卷数据丝毫不动，用指针来记录这些重定向的地址。在利用RoFW方式做了快照之后，针对随后的每个针对这个卷的上层IO，程序都需要查表确认是否需要重定向到新卷（或者本卷为快照所保留的空间），这种做法对性能是有较大影响的，为此，有人发明了另一种方式来保存快照数据。

快照生成之后，如果上层有针对原卷某个或者某些自从快照之后从未未被更新过的LBA块的写IO请求，则在更新这些LBA扇区之前，先将原来扇区的内容复制出来，放入一个空闲卷，然后再将新数据写入原卷。也就是说，旧数据先占着位置，等什么时候新数据来了，旧数据再让位，一旦原卷某个LBA的块在快照之后被更新过了，则以后再针对这个LBA块的写IO，可以直接覆盖，不需要提前复制，因为第一次更新此块的时候已经将原块数据复制保留了。这样，原卷上的数据随时都是当前最新的状态，所以针对快照之后的每个上层IO，不必再遍历映射表，直接写向原卷对应的地址，如果是写入一个快照之后从未被更新过的块，则需提前将原块复制保留，这种方式称为**CoFW**（其实应当叫做 **Copy on First Write**，见下文），写前复制。

在“照”下了这一时刻卷上的数据之后，为了保险起见，最好对那个时刻的数据做一个备份，也就是将快照对应的数据复制到另外的磁盘或者磁带中。如果不备份快照，那么一旦卷数据有所损毁，快照的数据也不复存在，因为快照与当前数据是共享LBA扇区的（如果没有更新原卷扇区的话）。

3. RoFW方式与CoFW方式比较

不管是CoFW还是RoFW，只要上层向一个在快照之后从来没被更新过的数据块进行写IO更新，这个IO块就要占用新卷上的一个块（因为要保留原块的内容，不能被覆盖），如果上层将原卷上的所有扇区块都更新了，那么新卷的容量就需要和原卷的数据量同样大才可以。但是通常应用不会写覆盖面百分之百，做快照的时候，新卷的容量一般设置成原卷容量的30%就可以。它是一个经验值，当然要根据具体业务场景来判断具体值。

CoFW方式下，快照生成之后，如果上层需要更新一个从来没有被更新过的块，则系统首先将这个源块读出，再将其写入到新卷，然后将更新的数据块内容覆盖写入到原卷对应的块，需要三步动作：一次读和两次写。RoFW方式下，同样的过程只需要一次写入即可，也就是将更新数据直接写入到新卷，同时更新映射图中的指针（内存中进行）。所以RoFW相对CoFW方式在IO资源消耗与IO延迟上有优势。

由于只是在首次覆盖写的时候才会发生Copy或者Redirect，那么如何区分某个写IO针对的块是首次被覆盖还是之前已经被覆盖过？这需要有一个记录表（文件级快照）或者位图（卷级快照）来记录每个块是否被覆盖过。

提示：对于卷级快照，可以使用位图来充当这个角色，因为源卷块地址是连续固定的，已经被覆盖过或者重定向写过的可以对应1；未被覆盖则对应0。针对每个写IO可以先查询这个IO的目标地址对应的位，如果为1则表示已处理过，则直接写入；如果为0则表示尚未被处理，需要先执行CoFW或者RoFW。

不管是CoFW还是RoFW模式下，对于每个上层写IO，由于都必须

遍历一下这个映射表（位图），以便确定此IO请求的LBA或者文件地址是否已被CoFW或者RoFW处理过，从而做出相应动作。而对于读IO，CoFW模式下由于源卷或者源文件系统总是代表当前最新的状态，所以任何读IO都会直接被下发到源来执行，也就是直接从源读出，而对于RoFW模式，则必须也查询这个映射表或者位图来查看读IO目标地址是否被处理过，如果是，则转向重定向之后的地址读；如果没有，则直接下发到源中来读出数据块。

RoFW会影响读性能。RoFW模式的快照生成之后，甚至将全部快照删除之后，不好清理战场，都会影响后续的所有读和写的性能，后遗症明显。因为重定向写出去之后，数据块的排布都是乱的，这样的话，就会严重影响后续读写性能。而CoFW模式下，源卷总是最新时刻的影像，删除快照之后战场自动清理，没有任何后遗症。

所以综合来讲，RoFW比较吃计算资源（后续的读操作由于连续变随机，也会很吃资源，有永久性后遗症），而CoFW比较吃IO资源。此外，还必须考虑一点，就是CoFW动作一般并不会永远都高频率地发生，对于一个特定文件系统或者卷，总有一些区域是热点，也就是应用程序总有趋势只频繁访问和更新某些地址，这与应用程序业务逻辑有关。

这样的话，当被覆盖第二次以及之后，CoFW模式就不会再发生IO惩罚，而读IO一直都没有惩罚，此时只剩下每次写IO时的映射表/位图遍历过程，会消耗一定的计算资源；那么对于RoFW模式，就算全部源文件系统或者源卷的数据块都被Redirect过了，那么虽然整个过程中不产生任何惩罚IO，但是针对每个读或者写IO，也就是每个IO，不管读写，均需要遍历映射表/位图，永远无法摆脱对计算资源的消耗。

尤其是对LUN卷级的快照下，原本卷在底层磁盘上的分布是很简单且定死的，所以寻址就非常迅速，但是RoFW模式的快照引入之后，LUN中的块会被随机的重定向写出到另外的空间，这样一方面需要记录比原本复杂得多的元数据指针链，降低了寻址速度（这一点对于文件系统级快照表现的还不是很明显，因为文件系统元数据链本来就很复杂）；另一方面这些被写出的块不一定是按照与源LUN相同的物理上连续排列的，这样在连续IO情况下便会产生严重性能下降（这一点不管是对文件级还是LUN卷级快照，影响都是很明显的，因为文件系统内的文件一般也是尽量物理上连续存放的）。

这一点相对于CoFW模式就逊色很多了。所以，绝大多数厂商还是使用CoFW方式来做快照。但是对于一些本来就使用LUN随机分块分布模式的存储系统比如NetApp和IBM的XIV，它们使用的就都是RoFW模式。显然，原本其LUN的元数据链就很复杂，再加上原本就是一种随机分布，所以RoFW的后遗症对于它们来讲反而是正常现象了。

此外，不管是CoFW还是RoFW模式，比如某个IO是64KB大小，那么此时你就要重定向这64KB或者CoFW出原来的64KB到另外空间，如果某时刻某IO是4KB大小，那么你就要去重定向这4KB或者CoFW出原来的4KB数据到额外空间，粒度不同，占据空间不同，元数据指针长度也就不同，这会导致算法更加复杂。

解决的办法是固定一个粒度，比如就用64KB，如果某个IO写为4KB，在RoFW模式下，则可以将这4KB目标地址所落入的那64KB的数据块读出，然后将4KB的新内容在内存中覆盖到这64KB中对应的地址上，然后再将更新后的这64KB重定向到另外的空间，以后再有针对这64KB目标块的写入，则直接覆盖写到重定向之后的空间；在CoFW模式下，则直接读出原来的64KB写到额外空间后，直接将新的4KB覆盖到

源卷对应地址，之后再针对源卷的这64KB地址范围内的写操作时，就无须再CoFW了（当然如果又做了一次快照，那就另当别论了）。

这么做的优势是，剩余空间内可以以一个固定粒度来占据空间，而不是一会4KB，一会32KB无序地乱放，这样就可以降低元数据指针的复杂度和无序度，提高查询效率。相对没有什么劣势，虽然首次复制数据量增加，但是却可以简化后续每个IO都要复制所引发的持续性延迟升高，一次做完还是不紧不慢地做，这种情况下还是选择前者比较好。后文中会有详细的设计例子来论述这两种方式的区分。

卷级的快照，仿佛就是增加了一个“卷块映射系统”，其作用与文件系统大同小异，只不过文件系统处理的是文件名和块的映射关系，而“卷块映射系统”处理的是块与块的映射关系。后者的元数据比前者简单得多，也好处理得多，粒度也大很多。

4. 快照的意义和作用

提示： 快照所冻结下来的卷数据，无异于一次意外掉电之后卷上的数据。为什么这么说呢？

我们可以比较一下，意外断电同样是保持了断电所处时间点上的卷数据状态。我们知道，不管是上层应用，还是文件系统，都有自己的缓存，文件系统缓存的是文件系统元数据和文件实体数据。并不是每次数据的交互，都同步保存在磁盘上，它们可以暂时保存在内存中，然后每隔一段时间（比如某些版本的Linux系统默认为30秒），批量Flush到磁盘上。当然编程的时候也可以将每次对内存的写，都Flush到磁盘，但是这样做效率和速度打了折扣。而且当Flush到磁盘的时候，并不是只做一次IO，在数据量大时会对磁盘做多次IO。如果快照生成的时间恰恰

在这连续的IO之间生成，那么此时卷上的数据，实际上有可能不一致。

磁盘IO是原子操作（Atomic Operation），而上层的一次事务性操作，可以对应底层的多次原子操作。这其中的一次原子操作没有业务意义，只有上层的一次完整的事务操作，才有意义。所以如果恰好在一个事务操作对应的多个原子操作的中间生成快照，那么此时的快照数据，就是不完整的，不一致的。

文件系统的机制总是先写入文件的实体数据到磁盘，文件的元数据暂不写到磁盘，而是先保存于缓存中。这种机制是考虑到一些意外事件，如果FS先把元数据写入磁盘，而在准备写入文件实体数据的时候，突然断电了，那么此时磁盘上的数据是这么一个状态：FS元数据中有这个文件的信息，但是实体数据并没有被写入对应的扇区，那么这些对应的僵尸扇区上原来的数据便会被认为就是这个文件的数据，显然后果不堪设想。

所以FS一定是先写入文件实体数据，完成之后再批量将元数据从缓存中Flush到磁盘。如果在实体数据写入磁盘，而元数据还没有写入磁盘之前断电，那么虽然此时文件实体数据在磁盘上，但是元数据没有在磁盘上，也就是说虽然有你这个人存在，但是你没有身份证，那么你就不能公开地进行社会活动，因为你不是这个国家的公民。虽然文件系统这么做，会丢失数据，但是总比向应用提交一份驴唇不对马嘴的数据强！

实验：就拿Windows来说，首先创建一个文件，并在创建好的瞬间，立即断电，重启之后，会发现刚才创建的文件没了，或者复制一个文件，完成后立即断电，重启之后也会发现，复制的文件不见了，为什么？明明创建好的文件，复制好的文件，

为什么断电重启就没了呢？原因很简单，因为断电的时候，FS还没有把元数据Flush到磁盘上，此时文件实体收据虽然还在，但是元数据中没有，那么当然看不到它了。

总之，快照极有可能生成一份存在不一致的卷数据。既然这样为何还要使用快照呢？因为相对于停机备份，人们更接受使用快照来备份数据，即使快照可能带来数据不一致。但停机备份所带来的损失，对于某些关键应用来说是不可估量的，而快照只需要几秒钟即可完成，应用只需静默几秒钟的时间。文件系统或者卷擅自将IO先存放到队列中，等待快照完成后，再继续执行。然后可以随时将这份快照对应的数据复制出来，形成备份。

使用快照必须承担数据不一致的风险。也可以这么形容：快照可以让你不用“意外磁盘掉电”，就能获得一个时间点瞬间磁盘备份，虽然数据此时是不一致的。快照可以任意生成，而占用的空间又不会很大（随原卷数据改动多少而定），最重要的是，利用快照可以做在线快速恢复，只要快照没有删除，恢复也同样仅仅需要几秒钟时间，与快照生成的道理一样，不用停机。因为利用快照恢复数据的时候，只要在内存中做一下IO重定向，那么上层IO访问的，就立即变成了以前时间点的数据了。这是快照的作用之一，即快速恢复，不管源卷数据量有多大，即便是几TB，恢复到指定时间点状态也只需要数秒钟，因为无须复制过程，只需重定向MetaData。

快照可能不一致这个问题，也不是不能解决。既然快照无异于一次磁盘掉电，那么，利用快照恢复数据之后，文件系统可以进行一致性检查从而纠正错误。数据库管理系统也同样会利用其日志来使得数据文件最终处于一致状态。

另外，现在几乎所有的快照的解决方案，都是在主机上安装一个代理软件，当在存储设备上执行快照之前，代理软件会通知应用或者文件系统将缓存中的数据全部Flush到磁盘，然后立即生成快照，这样快照的一致性就得到了保护。快照的管理和创建也都是使用这个代理程序界面完成，代理程序会与存储设备进行通信以便传输指令和查询信息。除了文件系统代理之外，还有各种应用系统代理，比如Oracle、SQL Server等，这些代理在存储层触发Snapshot之前会与对应的应用程序通信通知应用程序将缓存中的内容写到磁盘，从而达到底层数据的一致性。如图16-5所示，左右侧分别为一个Oracle和SQL Server的代理程序配置界面。

图16-5 Agent配置界面

对于文件系统级的快照，应用程序可以直接在操作系统中看到快照，可以直接读取这份快照中的文件。而对于卷级的快照，只有存储设备自己可以管理，操作系统对此一无所知。

对卷做快照之后，一旦原先的数据块被更改/覆盖，则这些块要么在写入前被复制出来，要么原地锁住不动，新数据重定向写到空闲区域。总之，这两种情况下，系统内整体空余空间都要被额外数据占用而减小。

然而，如果对某个块进行二次更改/覆盖，则可以直接在这些新块之上进行操作而不需要写前复制或者再次重定向写入，除非在第一次更改/覆盖和第二次之间又创建了一个或者多个快照。所以CoFW其实应当叫做Copy On First Write更加准确一些。

提示： 如果将原来的数据块删除了，在已经做了快照的情况

下，系统将做出什么行为？删除原块对应的行为其实是在文件系统的inode tree中将对应的指针消除，表现为更改/覆盖对应inode所在的块，所以不管删除多大的文件，也只有对应的inode块被更改/覆盖，被删除数据依然存在于磁盘上。然而这些数据对应的块在当前活动文件系统下的簇位图中已经被标记为空闲块了，此时一旦又有新数据被追加写入到这些空闲块，那么依然会被重定向或者写前复制。

使用快照功能之前必须首先评估数据源IO行为，随着覆盖写入几率的增加，系统中可使用的空闲空间的数量也将随之增加。实际情况中，文件被覆盖写入的几率一般不高，比较高的是Create+Write、Delete、Rename、Open、Read、Truncate、Append等操作，其中Delete、Rename、Truncate操作会导致Snapshot占用额外空间，而这其中Delete和Truncate会导致被删除文件本身与其inode的block都占用额外空间，Rename只会导致对应inode block占用额外空间。Create+Write操作如果一旦覆盖了旧块，那么同样也会占用额外空间。

快照的另外一个非常重要的意义和作用，就是预防数据的逻辑上的损坏。所谓逻辑上的损坏，比如T1时刻对某卷做了Snapshot，T2时刻，管理员操作不当，误删除了此卷上的某个非常重要的文件，但是自己没有察觉。T3时刻，备份管理员对这个卷进行了全备份操作，并且之前的备份已经被抹掉。此时，这个非常重要的文件，看似是永久丢失了。其实不然，T1时刻，也就是这个文件尚存在于此卷的时刻，管理员做了一次Snapshot，则可以从这份Snapshot中将这个文件恢复。同理，如果T2时刻，此卷被大量病毒感染，中毒已深无可救药，Snapshot此时便是救命稻草，用Snapshot将这个卷恢复到T1时刻，几秒钟后，病毒全部消失。

另外，快照可以降低一致性备份的窗口，假如没有快照技术，如果想对某个卷进行一致性备份，则需要暂停上层应用程序的写IO操作，等待备份程序将这个卷的所有数据块全部复制完成之后，应用程序方可对此卷进行写操作。这显然是不可接受的，所以如果没有快照技术，单纯对卷进行一致性备份的话，备份窗口将会非常长。而如果对这个卷触发一份一致性快照（引入应用程序快照代理），然后立即允许上层写IO，之后在后台备份这份快照对应的所有卷数据，这样就会大大降低对应用程序的影像，而且由于无须考虑对当前系统IO的影像，还可以提高备份速度从而降低备份窗口。

其次，在备份完数据之后，如何检测这份数据是否真的是一致的？是否用其恢复之后，应用程序可以正常启动并且处理它们呢？如果没有快照技术，则要实现这个目的，我们需要先将备份的数据恢复到一个独立的物理空间上，然后将其挂载到另外一台机器，在这台机器上启动对应的应用程序对数据进行检测，费时费力。有了快照就不同了，快照生成之后，在备份这份快照数据的同时，还可以将快照直接挂载到另外的主机，避免了最慢的一步，也就是数据物理恢复导入的过程，大大节约了时间和成本。

提示：目前几乎所有厂商的存储产品都可以实现快照，这似乎成了一个不成文的行业标准，不能实现快照的产品无法在市场上生存。目前市场上也有很多基于NTFS文件系统的第三方独立快照管理软件，例如“还原精灵”等，它们都是非常优秀的基于NTFS文件系统的快照管理软件。这些软件的作用方式一般是将自己的程序入口写入MBR，系统引导初期会首先加载这些程序，开机之后始终在操作系统下层运行，这样就可以肆无忌惮地接管操作系统对NTFS文件系统的所有操作。在这个

基础上，程序可以实现文件系统的快照以及管理这些快照。

5. 如何管理和使用快照

以上论述了Snapshot的原理和作用，本节我们将阐述一份Snapshot是如何具体被使用的，即生成了Snapshot之后，系统如何管理和使用它们。

我们知道，Snapshot是文件系统或者卷管理系统的映射指针链在某时刻的存根。现在的存储系统一般都支持对某个卷做多份Snapshot，比如16份、256份，IBM公司的XIV产品甚至可以超过60000份。每个厂家实现Snapshot的方式都不尽相同，底层对应的数据结构也不同，但均需要维护指针映射关系。

比如，有的厂家在文件系统总入口处设立多个Slot用以存放二级指针以便指向所有Snapshot的一级入口，Slot的数量决定了可以保存Snapshot的数量，顺着每个Snapshot的入口遍历下去就会得到整个文件系统的MetaData。有的厂家则在每个inode中增加了Slot来表明这个inode属于哪些Snapshot，整个文件系统只有一个入口，通过inode中的这些Slot来遍历所有Snapshot。有的厂家则尽量简化结构，用带外的方式创建一个独立的映射关系表，一旦某个Block将被覆盖，则将对应的inode等MetaData信息复制到这张表中随时备查。各家的实现和管理Snapshot的方式，见仁见智。

Snapshot对外表现为一个虚拟的卷，一般可以在存储系统的管理界面中看到，比如物理卷图标下面列出所有Snapshot卷，或者用命令行方式来列出Snapshot，比如，“`snaplist volume1`”，当然命令只是乱猜一个罢了。每当点开图形界面或者用命令行来查看Snapshot列表的时候，命

令的执行会触发存储系统检索整个文件系统的Snapshot链来查询当前Snapshot的情况，并将结果向外输出。所以每次查看Snapshot列表，是需要耗费一点时间的，通常不会很慢，延迟在几秒钟之内。图16-6所示为通过运行在主机上的代理程序所列出的Snapshot列表。

图16-6 快照列表

生成的Snapshot有两种使用方式，各个厂家在使用方法上大同小异。一种是直接在存储设备上利用某卷某时刻的Snapshot恢复该卷（比如命令“`Snaprestore snap1 volume1`”）。当然，恢复的时候一定要将这个卷unmap，或者在主机端将其unmount、varyoff或offline。否则主机端正在使用的情况下，存储端“擅自”做Restore之后，主机端文件系统缓存内的数据会覆盖掉restore之后的数据，导致FS不一致。Restore之后，主机重新挂载这个卷，看到的就会是以前的Snapshot对应的内容了。整卷Restore目前来讲用的不太普遍，毕竟发生整卷级别数据丢失或者逻辑错误的情况还是少数。用的最多的是第二种方式，即主机单独挂载某卷某时刻的Snapshot，恢复其中某个或者某些文件。

如图16-7所示，在代理程序界面中有Disk1的30个Snapshot，右击25，从弹出的快捷菜单中单击Mount Snapshot命令，之后Mounted状态栏显示“Yes”，表明这个Snapshot已经被成功地Mount到了主机端。

图16-7 挂载快照

如图16-8所示，此时，打开“我的电脑”，会出现一个新挂载的盘符，其中的内容便是Disk1在对应时刻的快照中的内容。找到需要恢复的文件，将其复制到源卷即可。完成之后，在代理界面中将其Umount即可。被挂载的快照卷，只是一份幻像，因为Snapshot中的内容与源卷

是息息相关的，一旦源卷发生底层错误，则其上的Snapshot也保不住，所以，将Snapshot挂载之后，可以用备份软件备份下来用以保存实体数据到另外的介质，比如其他磁盘，或者磁带。

图16-8 新盘符

同理，可以挂载所有的快照到主机，而不会影响源卷的读写访问。当然，这些快照此时都是只读的。要想得到一份可读并且也可写的Snapshot，由另一种技术来实现，叫做“卷Clone”。

6. 一些针对快照的高级优化技术

快照技术已经并不是多么高端的技术了。近年来，国内多家科研院所或者企业都在着力研究如何提高快照效率，降低快照生成之后源卷的写惩罚，也已经有多项专利登记在案。

比如某专利，其提供了一种快照空间的动态扩容方法，首先在逻辑卷的卷组空间中分配一快照空间，以创建快照，并且在逻辑卷有数据写入请求时，比对快照空间的总容量与快照空间的当前保存数据量，以得到表示快照空间的剩余容量大小的差值，检查差值是否小于一默认值；当差值小于默认值时，从卷组空间中划分一容量空间，以通过由标准扩容算法对快照空间进行容量扩展。

比如某专利，其提供了一种缩短写时复制快照写响应时间的方法，预先将存储系统的整个存储空间分为源数据区和快照区，源数据区存放主机访问请求的数据，快照区存放需快照保存的数据；设定快照块大小，构造快照链表头和快照索引结构根节点，快照链表保存所有快照元数据，快照索引结构用于判断相应数据块是否已保存在快照区；当接收

到主机发送过来的写请求时，顺序进行下述步骤。

（1）根据设定的快照块大小，将写请求数据依据其起始地址和长度进行分块；

（2）以每个分块号为关键字在快照索引结构中进行查找，若已存在，转步骤（8），否则顺序进行；

（3）判断快照区是否有足够的剩余空间，是则转步骤（5），否则顺序进行；

（4）自动删除快照区中存在时间最久的一个快照，并更新快照区大小，转步骤（3）；

（5）启动快照数据读进程，将需快照保存的数据块从源数据区中读出，并加入到快照数据写入队列尾；

（6）更新快照区内可用快照区大小和下一个可用扇区的位置；

（7）判断所有分块是否都已处理完毕，是则顺序执行，否则转步骤（2）；

（8）将主机发送的写请求数据写入源数据区；

（9）判断是否满足预先设定的快照数据写进程启动条件，是则启动快照数据写进程，将内存中积攒的快照数据写入队列中的数据写至快照区，否则顺序执行；

（10）结束。

7. 快照的底层架构设计实现详述

为了让读者尽可能深入地理解Snapshot的底层工作原理，作者在此自我演绎了一种Snapshot的实现方式，下面就给出架构设计思想说明，也欢迎广大读者与我交流探讨。以下所有设计概要均为作者个人演绎，如与某专利或者某厂商设计雷同，则纯属巧合，请勿对号入座。

1) 基于Copy On First Write的架构设计

本设计包含4个实体数据结构。第一个是用于存放变化IO块的IO仓库，可以是任何形式的高性能永久存储空间，再加上一份IO仓库的bitmap用来标记仓库中空闲与非空闲块。第二个是地址映射表，用于保存指针，地址映射表也存放于IO仓库的特定位置，并且每个快照各保存一份。由于考虑到内存占用和处理效率问题，地址映射表中不可能为每个针对源卷的LBA地址都有一个Slot，而只能是系统生成条目时的时候，利用二分查找方式将新条目插入表格中并且保证所有条目按照LBA地址大小的顺序排列，从而方便后续的步骤中对表中地址的查找，不需要做索引等其他结构，以降低系统资源耗费。第三个结构是RoFW数据映射表，这个表保存所有针对虚拟快照影像卷的写IO（可以参考卷Clone一节），同样是每个快照一份，只读挂载的快照则不需要这个数据结构。Bitmap中每一位可以表示一个LBA地址，也可以表示一大段LBA地址，比如4KB/16KB甚至128KB的Block，加大表示粒度可以大大提高处理效率和降低内存占用，但是在CoFW操作时就会浪费存储空间。比如，某个写IO只有4KB大小，但是也必须CoFW出这个IO目标地址段所落在的整个Block，但是后续写IO如果再落在这个Block，则无须CoFW操作了。粒度大小需要根据情况综合选择。

下面给出具体步骤演绎。

快照生成及生成之后的系统动作流程

(1) T0时刻，有卷V1，IO仓库V2。此时系统触发一份Snapshot。进入触发流程之后，系统创建一份针对T0时刻的快照S0的地址映射表F0，这份表在T0时刻为空，如图16-9所示。

图16-9 T0时刻

(2) T1时刻，有写IO将源卷LBA0地址上的A0更新为A1，系统首先检查所有已存在的地址映射表中哪个或者哪些没有针对源卷LBA0地址的映射条目，发现没有，所以CoFW出A0，将A0写入IO仓库的LBA0，然后在地址映射表F0中更新地址映射条目：LBA0=LBA0，意思是源卷在T0时刻的LBA0这个地址上的数据目前保存在IO仓库中的LBA0地址上。最后系统将A1正常写入源卷对应地址，如图16-10所示。

图16-10 T1时刻

(3) T2时刻，有写IO将源卷LBA30地址上的C0更新为C2，系统首先检查所有已存在的地址映射表中哪个或者哪些没有针对源卷LBA30地址的映射条目，发现没有，所以CoFW出C0，将C0写入IO仓库的LBA1，然后在地址映射表F0中更新地址映射条目：LBA30=LBA1，意思是源卷在T0时刻（T0时刻与T2时刻源卷LBA30地址上的数据相同）的LBA30这个地址上的数据目前保存在IO仓库中的LBA1地址上。最后系统将C2正常写入源卷对应地址。同时，系统再次触发一份快照S2，并为S2创建一份空的地址映射表F2。如图16-11所示。

图16-11 T2时刻

(4) T3时刻，有写IO将源卷LBA0地址上的A1更新为A3，系统首

先检查所有已存在的地址映射表中哪个或者哪些没有针对源卷LBA0地址的映射条目，发现在针对快照S2的地址映射表F2中没有对应的条目，所以CoFW出A1，将A1写入IO仓库的LBA2，然后在所有不存在LBA0映射条目的映射表中加入新地址映射条目：LBA0=LBA2，意思是源卷在T2时刻的LBA0这个地址上的数据目前保存在IO仓库中的LBA2地址上。最后系统将A3正常写入源卷对应地址，如图16-12所示。

图16-12 T3时刻

（5）T4时刻，有写IO将源卷LBA11地址上的B0更新为B4，系统首先检查所有已存在的地址映射表中哪个或者哪些没有针对源卷LBA11地址的映射条目，发现F0和F2都没有，所以CoFW出B0，将B0写入IO仓库的LBA3，然后在地址映射表F0和F2中同时更新地址映射条目：LBA11=LBA3，意思是源卷在T0或T2时刻（T0、T2和T3时刻LBA11地址的数据是相同的）的LBA11这个地址上的数据目前保存在IO仓库中的LBA3地址上。最后系统将B4正常写入源卷对应地址，如图16-13所示。

图16-13 T4时刻

（6）T5时刻，有写IO将源卷LBA11地址上的B4更新为B5，系统首先检查所有已存在的地址映射表中哪个或者哪些没有针对源卷LBA11地址的映射条目，发现F0和F2都有，所以系统不进行CoFW过程，直接将B5正常写入源卷对应地址，如图16-14所示。

图16-14 T5时刻

（7）任何时刻，有读IO针对源卷任何地址，系统将直接从源卷读取对应地址的数据返回给请求者。

客户端挂载某份快照进行IO读写时的系统动作流程

(1) T6时刻，系统接收到客户端请求将快照S0对应的T0时刻的源卷的虚拟影像映射给客户端。系统接收到请求之后，立即根据源卷的大小等属性，通过底层传输通道报告给客户端主机一个虚拟卷。

(2) T7时刻，主机识别到了这个虚拟卷影像（此时主机对源物理卷依然在进行读写IO，无影响），并且将其挂载到某个盘符下。此时有某个程序（比如文件系统）尝试读取这个卷的LBA11地址对应的数据。

(3) T8时刻，系统收到这个读IO请求之后，立即扫描针对快照S0的地址映射表F0，判断对应LBA11的Slot是否被记录了对应的IO仓库存储空间的映射LBA地址。发现LBA11对应的Slot存在地址映射，所以立即将映射条目中右侧也就是IO仓库的对应地址提取并且读出IO仓库中对应这个地址（本例中为LBA3）的数据，这个被读出的块就是T0时刻快照S0所对应的实际数据块了。

(4) T9时刻，系统将读出的数据块返回给客户端主机。

(5) T10时刻，系统接收到来自客户端的针对源虚拟影像卷LBA5的读请求，立即扫描映射表F0发现源卷LBA5的Slot中并没有映射条目，所以系统立即从源卷中读取LBA5地址对应的内容并返回给客户端主机。

说明： 有一点必须说明，查询线程在查找映射表之前，必须先向CoFW线程查询当前时刻是否存在任何针对源物理卷的写IO正在被执行，如果有，IO的地址是多少，如果地址恰好就是需要查询的地址，则查询线程必须先等待CoFW线程完成CoFW操作之后（此时CoFW会向映射表中更新刚才这个CoFW

的数据在IO仓库中的地址），再去查询映射表中对应的条目，此时就可以查到最新的结果了。并且，如果查询线程查询的结果是直接从源物理卷中读取内容，则在查询线程完成读取之前，CoFW线程不能进行针对这个地址的CoFW操作。如果系统不这样做，也就是不与CoFW线程交互，那么查询线程很有可能从源卷中读取的是最新刚被写入的数据而不是对应以前快照时刻的数据。

（6）T11时刻，系统接收到主机客户端针对虚拟影像卷任何一个地址的写IO操作，则系统立即将这个写IO和其对应的LBA地址一同存放在另外一个单独的数据映射表中，与地址映射表一样，每个条目Slot中的LBA地址按照大小顺序排列，只不过这个Slot中存放的是虚拟影像卷的LBA地址和对应的主机写IO实体数据（当然也可以将实体数据放到公用IO仓库中然后在这个表中存放数据在仓库中的LBA地址指针，视设计不同而定）。一旦第一个写IO被系统接收之后，针对随后的任何一个客户端发送的读或者是写IO，系统都要首先查询这个RoFW数据映射表，如果找到对应的映射，则直接将表中对应的数据返回给客户端；如果没找到对应条目，则再执行与上几步中同样的动作。上一步中的线程依赖关系同样要考虑。

（7）T12时刻，客户端要求挂载快照S2对应的虚拟影像卷。系统接收到请求之后，做与第（1）步相同的工作。如果接收到客户端主机针对S2影像卷的任何IO请求，则做与第（2）～第（5）步相同的动作，只不过系统要去扫描S2的映射表F2了。两个虚拟卷可以同时接收IO。

客户端决定将快照S0进行Rollback（Restore）时系统的动作流程

（1）T13时刻，客户端要求将快照S0直接回滚覆盖到源物理卷。系

系统接收到请求之后，直接向客户端返回已经成功Rollback（与直接挂载虚拟影像卷过程类似）。这里虽然也可以选择前台Rollback，即Rollback所有步骤完成之前，系统不接受针对S0快照点卷的任何IO操作。但是为了保证操作友好性，在此设计为后台操作。后台操作的结果是，主机直接认为Rollback成功，则可能立即向源卷发起读写IO。还需要注意的一点是，在Rollback之前，主机客户端一定要将其Buffer中的内容Flush到源卷，否则源卷一下子被Rollback，但是主机中却保存着Rollback之前的缓存，随后又将这些缓存Flush到Rollback之后的源卷，那么数据就不一致了。不一致的后果是可怕的。所以，在Rollback之前，干脆在主机端Unmount源卷，Rollback完成后再Mount回来即可。

（2）T14时刻，系统开始执行与挂载虚拟影像读写时相同的动作，但是在后台，系统会将原先CoFW出来的数据块以及刚刚被RoFW（如果后台操作未完成时客户端有写IO进入）的数据统统覆盖到源物理卷对应地址上，在读取IO仓库中的CoFW块时要首先查询RoFW数据映射表中是否已经针对当前操作的地址有被写入的数据；如果有，抛弃原来被CoFW的数据转而将刚被更新过的数据写入源卷（在处理客户端写IO请求时，也可以直接将写IO数据覆盖到源物理卷对应地址，无须RoFW，节约一轮IO操作，但必须记录每个在Rollback开始与完成之间发生的写IO地址以便在Rollback时跳过这些地址，可以用Bitmap来记录）。同样，线程依赖关系必须考虑。处理完的CoFW块将在IO仓库的Bitmap中被标记为空闲以便腾出空间。

说明： 整个步骤中需要着重注意的是，比如T5时刻系统状态图所示，当系统要将C0覆盖C2的时候，此时由于C2为快照S2的基准块，所以，如果选择保存S2快照，那么依然需要将C2先CoFW之后放入IO仓库，然后更新F2映射表中对应的条目。

即，在做覆盖操作之前，必须参考恢复快照点之后所有快照点的地址映射表，一旦任何一份表中对应的地址没有映射条目，则依然需要CoFW过程。

然而，目前市面上的产品几乎都严格遵照历史规则，即，既然选择了回滚到T0时刻的S0快照点，那么T0时刻之后所发生的所有事件就不应该存在，所以S2也不能存在，所以大部分实现案例中，在选择恢复到某个快照点的时候，都会同时将这个快照点时间之后的所有快照删除。但是我们这种设计是可以选择保存后面的快照点的。同理，恢复到S2快照点与恢复到S0快照点的具体动作流程相同。

（3）T15时刻，当地址映射表与RoFW数据映射表中的所有条目都被处理完成之后，系统立即将这两份表彻底删掉。此刻之后，就相当于S0快照没有被创建过，而系统只剩下了S2快照。随后的动作流程与生成S2快照之后的动作相同。

最后还需要明确一点，如果在整个后台过程完成之前，再有新的Snapshot被创建，则新Snapshot创建之后的CoFW过程将会与后台Rollback操作过程相冲突，除非引入额外的CoFW操作才能保证数据一致性，所花费的开销和动作流程与上文中的保存S2快照的开销和动作流程相同。当然，如果一开始就选择使用前台方式Rollback，则Rollback的时间和效率都会增加，只不过需要用户等待一段时间。

客户端决定删除快照**S0**时系统的动作流程（**S0**尚未被删除时，从**T13**时刻开始）

（1）T13时刻，客户端卸载了S0快照虚拟影像卷，系统收到卸载请求之后，将RoFW数据映射表全部删除，其他不做更改。如果客户端再

次挂载该快照，则会重新得到一份干净的T0时刻的卷影像，之前的写IO数据由于被系统删掉，所以全部丢失。如果客户端希望在卸载虚拟卷之后系统能够依然保持所做的更改，则需要通知系统创建一份针对快照S0的Clone卷，即针对S0的二次虚拟影像卷。所谓Clone卷也只不过就是系统会永久保存RoFW数据映射表，除非显式地收到删除请求。Clone卷下文也会有所介绍。

(2) T14时刻，客户端请求系统删除快照S0。系统接收到请求之后，首先扫描对应S0快照的F0地址映射表并扫描其他所有快照的地址映射表，找出只有S0映射表F0中存在而其他快照映射表中都不存在的对应条目，比如T5时刻的LBA30映射条目，从IO仓库bitmap中标记对应的CoFW块为空闲。当S0映射表中所有条目都扫描并处理完成之后，删除整个S0映射表F0，此时S0快照便删除完毕。这个过程中依然涉及线程依赖关系，需要考虑。另外，这个过程可以前台操作，也可以后台操作，但推荐后台操作，这样对系统和随后的IO无影响。

(3) 删除任何一个快照，流程皆相同。

如何计算CoFW模式下两份快照影像之间所更改过的数据地址

有时候系统需要得知前后两份快照之间所变化的数据地址，这种情况在实现增量备份的时候很需要。比如A时刻做了一份快照并将其做了备份，B时刻又做了一次快照，此时没有必要将B时刻的快照也完整地备份下来，如果只备份B时刻快照相对于A时刻快照之间变化过的数据，则可以大量节省备份时间和空间。待恢复时刻A快照的时候，正常恢复，待恢复时刻B快照的时候，当恢复程序读取到更改过内容的地址时直接读取时刻B的增量备份中对应地址的数据即可。还有一种情况需要使用快照比对，那就是远程数据容灾系统的底层设计，诸如

EMC/Mirrorview、3PAR/RemoteCopy、NetApp/SnapMirror等远程数据复制软件均使用快照比对技术来向远程批量传送变化的数据，下一章将详细描述这些技术的底层实现方式。

对于CoFW模式的快照，比较任意两份快照之间所不同的数据只需按照下面步骤操作即可

(1) 某系统先后生成了3份快照，S0、S1、S2。现需要比较以得知快照S0与S2之间的变化数据。系统首先对比S0和S1，找出在S0地址映射表中存在但是在S1地址映射表中不存在的地址映射条目（先忽略映射结果的异同，即条目右侧的地址。只要某个地址存在条目就是存在，不存在条目，就是不存在），以及S0中和S1中都存在但是映射结果不同的地址映射条目。将所有条目收集到一份新列表中。

(2) 之后，系统再按照相同的规则比对S1和S2的地址映射表，将收集到的地址追加到刚才的新列表之后。

(3) 对新列表中的地址条目按照源卷地址进行排序。

2) 基于RoFW模式的快照设计

RoFW模式一般没人用在原生快照设计中，除非是架构迥异的快照实现方式比，如NetApp WAFL快照（我们将在后文中简要介绍WAFL快照的实现原理）。究其原因是因为RoFW模式实现起来相对CoFW方式复杂。但是由于其对写性能的影响显著低于CoFW方式，在这里也简要地演绎一下这种模式。

快照生成及生成之后的系统动作流程

(1) T0时刻，有卷V1，IO仓库V2。此时系统触发一份Snapshot。

进入触发流程之后，系统创建一份针对T0时刻的快照S0的地址映射表F0，这份表在T0时刻为空，如图16-15所示。

图16-15 T0时刻

(2) T1时刻，有针对源卷LBA0地址的写IO数据A1进入，此时系统直接将A1写入IO仓库中的LBA0存放。并在S0对应的地址映射表F0中的LBA0的Slot中加入映射关系“LBA0=LBA0”如图16-16所示。左边是源卷中的地址，右边是IO仓库中的对应数据存放的地址。这里的意义与CoFW方式不同了，这里的意思是，源物理卷地址LBA0的最新数据内容在IO仓库中的LBA0地址存放而不是源卷自身，源卷自身永远都是S0时刻的影像，永久冻结，拒绝写入，除非所有快照都被删除。

图16-16 T1时刻

(3) T2时刻，有针对源卷LBA30的数据C2进入，系统依然直接将C2写入IO仓库的LBA1地址存放，并在映射表F0中对应LBA30的Slot中增加映射条目。最后，系统收到命令触发一份新快照S2，系统会为S2也生成一份映射表F2，如图16-17所示。

图16-17 T2时刻

(4) T3时刻，再次有针对源卷LBA0地址的数据A3进入。此时，由于第二份快照S2已经生成，而S2的生成会引发系统动作的很大变化。此时，源物理卷、F0映射表以及表中的指针所指向的IO仓库中的数据，共同组成了快照S2的基准卷，这三者的共同体也就是源卷在T2时刻的影像，即快照S2，也将被永久冻结，拒绝写入，如图16-18所示。

图16-18 T3时刻

系统就像上几步一样，直接将A3写入IO仓库中的新地址LBA2中（LBA0和LBA1被冻结但是其他地址未被冻结），并且在F2映射表中将LBA0的Slot写入映射指针。

（5）T4时刻，再次有针对源卷LBA0地址的数据A4进入。此时，系统检查最晚也就是最后一份快照映射表中是否有对应这个地址的映射条目，如果有，则提取出映射的IO仓库地址同时将数据写入IO仓库中对应的地址，也就是覆盖了IO仓库中原来这个地址上的数据。本例中，系统发现存在LBA0的条目，所以直接用A4覆盖了IO仓库中原来的数据A3。映射表中不做任何更改。如图16-19所示。

图16-19 T4时刻

（6）T5时刻，有针对源卷地址LBA11的IO写入数据B5进入。此时系统依然首先检查最晚也就是最后一份快照映射表中是否有对应这个地址的映射条目，如果没有，则将数据写入到IO仓库中的一个新地址中。本例中，F2表中没有LBA11的映射条目，所以系统直接将B5写入了IO仓库中的LBA3中，并在F2表中更新了条目，如图16-20所示。

图16-20 T5时刻

（7）T6时刻，有针对源卷LBA0地址的读操作进入。此时系统检查最晚的一份快照中是否有LBA0地址的映射条目，如果没有，则继续查找前一份快照是否有对应地址的映射条目，如果都没找到映射条目，则直接从源物理卷中对应地址读出数据并返回给客户端。如果按时间倒序的顺序查询之后最终在某份快照映射表中查到了对应地址的条目，则提取映射的IO仓库地址，并从IO仓库中将这个地址对应的数据读出并返回给客户端。本例中，系统在最晚的快照映射表F2中找到了LBA0的映射

条目，立即从IO仓库中将LBA2处的数据读出，返回给客户端。经过上述的算法，才能保证传给客户端的数据是最新的，而不是某份快照时间点对应的之前的数据。

（8）T7时刻，系统再次触发一份快照S7，创建了对应的F7映射表。此时，F0和F2均都处于冻结状态，如图16-21所示。

图16-21 T7时刻

（9）T8时刻，有针对源卷的LBA0和LBA17以及LBA5的三个写IO先后同时进入。系统按照规则，查询最后一份快照地址映射表也就是F7中是否有针对这三个地址的映射条目，发现都没有，所以将这三个IO数据写到IO仓库中的空闲位置，然后在F7中更新映射条目，如图16-22所示。

图16-22 T8时刻

客户端挂载某份快照进行IO读写时的系统动作流程

（1）T9时刻，系统接收到客户端请求将快照S0对应的T0时刻的源卷的虚拟影像映射给客户端。系统接收到请求之后，立即根据源卷的大小等属性，通过底层传输通道报告给客户端主机一个虚拟卷。

（2）T10时刻，主机识别到了这个虚拟卷影像（此时主机对源物理卷依然在进行读写IO，无影响），并且将其挂载到某个盘符下。此时有某个程序（比如文件系统）尝试读取这个卷的LBA0地址对应的数据。

（3）T11时刻。RoFW模式的原生快照的好处是，把复杂留给后台，把简单留给前台。确实是这样的，写的时候直接写入对应位置，不需要CoFW方式中的多余的一读一写操作，只不过写之前需要做一些映

射运算以判断是覆盖还是新写到空闲位置，但是运算过程总比IO过程快。同样，在读取系统最早的也就是第一份快照虚拟影像的时候，也是非常痛快淋漓的。本例中挂载了S0时刻快照，也就是最早的第一份快照，上文中说过，S0快照其实就是当前的物理源卷，因为自动S0生成之后，源卷就被永久冻结了。所以此刻，系统针对客户端发起的读请求将直接定向到源卷对应地址，不牵扯任何映射运算过程。

（4）T12时刻，系统接收到客户端请求将快照S2对应的T2时刻的源卷的虚拟影像映射给客户端。系统接收到请求之后，立即根据源卷的大小等属性，通过底层传输通道报告给客户端主机一个虚拟卷。

（5）T13时刻，主机识别到了这个虚拟卷影像（此时主机对源物理卷依然在进行读写IO，无影响），并且将其挂载到某个盘符下。此时有某个程序（比如文件系统）尝试读取这个卷的LBA0地址对应的数据。

（6）T14时刻。上文中描述过针对最早快照虚拟卷的读过程。本步中针对的则不是最早的快照，而是随后的快照。系统收到针对虚拟影像卷LBA0地址的读请求之后，会查找当前被挂载的快照点的上一个快照点对应的地址映射表。这里需要搞清楚，RoFW模式的快照实现设计与CoFW实现方式在这个步骤中的动作迥异。也就是说，本次快照的影像其实是由源物理卷+上一份快照对应的地址映射表所共同冻结而成的，而最后一份快照地址映射表中的指针表示的则是源物理卷的最新数据状态，而不是这份快照时刻的状态，不要搞乱。本例中，系统查询F0表中是否有LBA0地址的映射条目，如果有，则提取映射地址并从IO仓库中读取对应地址的数据并返回给客户端。

（7）T15时刻，系统接收到针对S2快照虚拟影像卷LBA11地址的读操作。系统依然去查询前一份快照也就是S0对应的地址映射表F0，本例

中，没有找到LBA11对应的映射条目，则系统将会继续查前一份快照映射表中对应地址的映射条目，直到找到为止。本例中，已经到了尽头，所以系统直接从源物理卷中读取LBA11地址对应的数据并返回给客户端。

说明： 值得说明的一点是，RoFW设计模式下，不像CoFW模式下存在线程依赖关系。对快照虚拟影像卷的读操作并不会与对源卷的写操作之间有任何牵连和影响。

（8）T16时刻，系统接收到针对S2快照虚拟影像卷LBA0地址的写入请求。与CoFW的处理方式相同，此时系统将新创建一个针对S2快照的RoFW数据映射表，并将待写入的数据写入表中对应地址的Slot。在随后的读请求处理过程中，要优先查询RoFW表，过程与CoFW过程相同，其他步骤不再重复论述。

客户端决定将快照S0进行**Rollback（Restore）**时系统的动作流程

（1）T17时刻，客户端要求将快照S0直接回滚覆盖到源物理卷。系统接收到请求之后，直接向客户端返回已经成功Rollback（与直接挂载虚拟影像卷过程类似）。这里虽然也可以选择前台Rollback，即Rollback所有步骤完成之前，系统不接受针对S0快照点卷的任何IO操作。但是为了保证操作友好性，我们在此设计为后台操作。后台操作过程中如果遇到客户端写IO，处理方式与CoFW模式下的处理方式相同，见前文。

（2）T18时刻，系统开始在后台进行Rollback操作。由于S0快照点其实就是当前的源物理卷，所以Restore过程不需要任何额外操作，直接就完成了。要注意的是，S0对应的F0和S2对应的F2映射表需要保留，

前提是客户端想保留S2和S7时刻的快照的话，因为S2快照需要参考F0，S7要参考F2。但是F7映射表无须保留，因为F7中的指针表示的源物理卷的最新状态变化指针，既然选择回滚，那么就代表最新时刻的数据已经不需要了。

一般是不推荐选择保留晚于S0的其他所有快照的，因为保留这些快照需要支付额外的系统资源。如果客户端明确选择保留，则系统会将S0之后的所有地址映射表封存到一个特定的位置，永久冻结（这里有一个规律，即如果用户选择保留的所有快照中最晚的一个是SX（ $X > S0$ ），则映射表FX及其之后的所有映射表皆可删除）。然后系统开始新一轮的快照周期，即必须首先再生成一个S0'快照，创建一份新的F0'空映射表，然后再接受客户端的读写IO操作，随后生成的快照也按照顺序重新编号。如果后来用户选择要Rollback或者挂载之前生成的某份快照，则系统会调出封存的那几份映射表，根据永久冻结的源物理卷，一层层累加从而获得对应时刻快照的影像。

如果客户端在Rollback的时候选择了不保存S0之后的快照，则系统会直接将F0及以后的所有F映射表全部删掉，映射表所对应的IO仓库中的对应地址块也全部标记为空闲。此时，源卷彻底被恢复到T0时刻状态，由于当前系统中已经没有任何快照存在，所以针对客户端的写IO操作，系统会直接写向源卷而没有任何RoFW操作，直到有又一个快照生成为止。

Rollback完成后，客户端针对Rollback之后的卷进行写操作时，其实依然需要RoFW到IO仓库中，至于写到IO仓库哪个位置（覆盖还是新写），就需要系统根据最后一份地址映射表判断。读和写操作均与之前的相关步骤相同。整个系统的源物理卷，只要还存在任何一个快照，则永远都是第一份快照被创建时候的状态，永久被冻结。

客户端决定将非**S0**快照**S7**进行**Rollback**（从**T17**时刻开始）时系统的动作流程

（1）**T17**时刻，客户端要求将快照**S7**直接回滚覆盖到源物理卷。系统接收到请求之后，直接向客户端返回已经成功**Rollback**（与直接挂载虚拟影像卷过程类似）。这里虽然也可以选择前台**Rollback**，即**Rollback**所有步骤完成之前，系统不接受针对**S7**快照点卷的任何IO操作。但是为了保证操作友好性，我们在此设计为后台操作。后台操作过程中如果遇到客户端写IO，处理方式与CoFW模式下的处理方式相同，见前文。

（2）**T18**时刻，若客户端选择不保留任何其他的快照，则系统首先将所有**F7**之前的地址映射表（不包含**F7**，**F7**将被删除）做合并操作，如果在多个表中遇到同一个地址的映射关系，则只留下最晚的表中的条目，被抛弃的条目对应的IO仓库中的地址会在bitmap中被标记为空闲。合并之后的新映射表命名为**F0'**。源物理卷+**F0'**共同组成了**T7**时刻快照**S7**时卷的影像，此时，由于系统内再无其他任何快照，所以系统扫描**F0'**中的每一条映射，然后将IO仓库中对应地址的数据读出并覆盖到源物理卷对应地址，全表处理完成后，IO仓库清零，**Rollback**完成，源物理卷上的实际数据就变为了**T7**时刻的影像，系统继续运作。

（3）**T18**时刻，若客户端选择要求保留**S2**快照，则系统首先将**F2**之前的所有映射表，本例中即**F0**永久封存，同时删除**F7**映射表（并在IO仓库bitmap中标记对应**F7**映射的所有块为空闲），之后，系统将待回复快照点之前的所有**F**映射表（本例中即**F2**和**F0**）按照上一步描述的规则做合并操作，合并之后的映射表保存为**F0'**，此时便可接受客户端读写IO操作，开始新一轮周期，**Rollback**完成。

(4) T18时刻, 假设系统有10个快照, 如果客户端选择恢复第8个快照, 同时保留第3个、第6个快照, 那么此时系统首先将第1个快照对应的映射表合并到第2个快照对应的映射表, 合并之后封存(为3号快照的影像指针), 然后将刚才封存的合并之后的映射表与第3和第4个快照对应的映射表再次合并到第5个快照映射表中, 合并后封存(为6号快照的影像指针)。删除F8映射表。然后, 系统将1~6号快照映射表与7号映射表合并, 合并后存为F0'映射表, 源物理卷+F0'共同组成了T8时刻快照S8时卷的影像, 最后打扫战场, 将IO仓库中由于映射表合并而产生的空闲数据块在Bitmap中标记为0, 即空闲。系统继续运作。

可以判断出来, RoFW模式的原生快照其流程过于复杂, 实现起来流程较多, 尤其是快照生成之后的读IO过程和快照挂载或者回滚之后的一系列流程, 都涉及到复杂的运算, 所以目前大部分产品都使用CoFW模式进行快照设计。

NetApp与IBM XIV存储系统使用的都是基于RoFW模式的快照实现模式, 选择这种模式与其底层设计息息相关。在后面章节会讨论NetApp为何会使用WR模式。在第15章会讨论IBM XIV系统为何也会使用WR模式。

如何计算**RoFW**模式下两份快照影像之间所更改过的数据地址

对于RoFW模式的快照, 计算两份快照之间的数据变化地址要比CoFW模式下容易很多。设系统先后创建了S0、S1、S2三份快照, 要求比对S0和S2之间的变化数据地址。

(1) 由于WR模式的快照下, 快照S0其实就是源物理卷, 而快照S2时刻的状态其实被表现在前一份也就是S1地址映射表中。所以系统直

接合并S0和S1地址映射表，将其二者所有存在的条目合并，如果遇到某个地址有两个条目，则抛弃时间点较早的映射表中的条目，只保留时间点最晚的映射表中对应地址的条目。

(2) 将合并后的条目保存于一张新表中，并按照源卷地址进行排序。

其实，这个过程与删除某快照的过程类似。

提示：关于比对两份Snapshot中的变化数据地址，有更好、更高效的实现方法。上文中介绍的在两种快照模式下实现比对的设计都是基于后处理的，即快照已经生成，而且生成之前没有做任何处理。如果使用中处理，即在第一份快照生成之后，系统立即为源卷生成一份全0的空Bitmap，随后的写IO系统均将这个IO地址对应的位在Bitmap中置为1，一直到第二份快照生成之后的瞬间，系统首先将这份Bitmap封存，然后再次生成一份全0空Bitmap再次重复刚才的步骤。被封存的Bitmap中为1的位就代表了两份快照生成的间隔期间，所有针对源卷的数据变化地址。这种处理方式是目前厂商所采用的。

3) NetApp WAFL快照实现方式简述

首先，WAFL严格来讲是一个文件系统，彻底的文件系统，而不是一个卷管理系统。这个文件系统与其他文件系统一样，将底层的存储空间虚拟为文件，文件可以任意凌乱地存放于存储空间的各个位置，使用inode链条来组织所有的零散块。卷，即Volume，在WAFL下面其实也是一个文件，卷也可以被任意零散地分布于存储空间各个角落，只要WAFL有这个需要。了解Filedisk的人都应该理解WAFL的这种基本作用

机制。所以WAFL很灵活，既然卷是一个文件了，那么就可以对这个文件进行任意的删除、修改、扩充、缩减等一切针对普通文件可以做的动作。更加离奇的是，每个文件中又可以嵌套一个文件系统，即文件中的文件，文件系统再次将文件作为底层存储空间，在文件中再建立多个二级文件，这便是LUN了。WAFL进程需要同时管理底层物理存储空间和二次映射的文件虚拟空间。

其次，WAFL的文件系统根入口不是一个，而是每个存储空间保存256个根入口，从每个根入口进入，便可以遍历出整个文件系统的元数据链条。也就是说，WAFL可以在一个存储空间内存放256份文件系统影像，每一个影像便是一个快照。可以对物理空间，也就是Aggregate做快照，也可以对Volume做快照，因为每个Volume也有256个根入口。这256个入口中的第一个入口被称为“当前活动文件系统”，也就是Active Filesystem（AFS）。从这个入口进入之后遍历出的就是当前的存储空间内最新数据，一旦某时刻触发一份快照，则WAFL会将当前AFS的根节点入口向下一个空闲的入口复制一份，即完成了快照的生成。生成之后，当有针对AFS的写入操作时，WAFL将这些数据统统写入存储空间内空闲的块中，并同时更新元数据，元数据的写入一样也需要写入空闲块中，链条指针一层层地被修改，最终更新到AFS的根入口指针。

这样，上一份快照中的所有数据没有一个块被覆盖，从对应的根入口进入，便会遍历出那个时刻的完整存储空间影像。即便系统没有生成快照，WAFL也不会覆盖之前的数据，而总是写入空闲的块。这样做有个好处就是在空间内有大量空闲块时，WAFL可以肆无忌惮地写入而不需要非得按照被覆盖的块在空间内的排列顺序而对应写入，这样可以提高性能。但是随着空闲空间的减少，WAFL这种性能的提升会骤降。

提示： 关于快照虚拟卷或Clone卷挂载后系统识别冲突的问

题：由于很多操作系统的卷管理层都会对系统所识别到的磁盘写入一些识别信息以便标识和区别每一个卷或者磁盘。而某个卷或者磁盘的快照或者Clone中的这些识别信息与源卷是完全一致的，所以，当源卷已经被挂载到系统中，又挂载了Snapshot虚拟卷或者Clone卷之后，此时由于识别信息完全相同，系统会认为是相同的卷又被挂载了一次而卸下源卷从而挂载了快照卷，或者发现新卷与当前的源卷识别信息一致而不做任何动作。不管哪种应对方式，都会导致问题的发生。所以解决这个问题的关键还在于位于主机端的快照代理程序，快照代理程序在让存储系统映射某个快照之前，会扫描系统中当前的所有已挂载的卷的识别信息，然后生成一个不与它们冲突的新识别信息并通向存储系统发起写IO请求将对应的扇区中的识别信息修改，然后再按照正常顺序挂载，这样就不会产生冲突了。

8. 快照的生成对系统读写性能的影响

由于每生成一份快照，系统就需要做更多的CoFW操作，而写性能本身就是衡量一个存储系统性能的重要指标，快照的生成无疑是雪上加霜。凡是使用CoFW模式实现快照的存储系统，皆无法逃脱写性能降低的厄运。相比来讲，RoFW模式则能够逃脱这个厄运。图16-23和图16-24所示为EMC Clariion系统的写IO性能影响统计图。

图16-23 1个快照 图16-24 8个快照

可以判断，当一个快照生成之后，系统的写IO响应值立即升高到一

个峰值，其原因是受到不断的CoFW的影响。此后的时间内，系统写IO响应逐渐降低到原来的水准，其原因是因为系统在对已经CoFW过的地址进行再次或多次写的时候，会直接写到源卷而不再进行CoFW动作。图16-24显示了生成了8个快照的系统写IO响应速度随时间的推移而变化的情况。可以看到每多生成一个快照，IO响应时间就比之前的最高响应时间再上升一些，这也是理所当然的，快照数量越多，处理就越慢。

快照的生成不仅对写性能有影响，对于RoFW模式的快照，其对读也一样有影响。由于数据块被不断重定向到其他空间，这就导致本来物理上连续的数据块在被重定向写出去之后可能就变得不连续了，甚至非常随机，那么此时针对这些地址的读操作就会受到影响。

鉴于快照的这些后遗症，在使用快照对系统进行瞬时恢复后，可以考虑在后台将快照中的数据回拷一遍，让其变得连续，从而恢复之前的性能。

16.2.4 卷Clone

1. 什么是卷Clone

顾名思义，Clone（克隆）是指源数据集某时间点的一份或者几份实实在在的实体复制。快照类似一个某时刻定型的影子，而克隆则是某时刻定型的实体。使用快照技术来制作克隆，先将影子定型，之后再将实体填充到影子当中。也就是说，首先对某个源数据集（源卷或者源文件系统）创建一份快照，之后将这份快照指向的所有数据块复制出来到一个额外存储空间，这样，被复制出来的所有数据就组成了源数据集在那个时刻的一个克隆实体。

我们知道Snapshot一般都是只读的，即这个“幻像”你只能去看，而不能被改变，就像历史一样。但是在计算机的世界里，这个结论是不成立的，Snapshot一样可以写。可能有人不太理解，Snapshot作为一个源卷在某一时刻的照片，其内容皆为源卷内容的投射，自身并不存在任何实体内容，向Snapshot写入数据，到底是个什么概念呢？正如Snapshot自身一样，可写的Snapshot也只不过是对指针的处理而已，即系统增加了一个RoFW数据映射表。比如，存储系统将Snapshot1映射给了HostA，HostA将其挂载到了F盘，某时刻HostA对F盘的第一个扇区写入全0，存储系统收到这个请求之后，会将这512B的数据重定向地写入一个空闲存储空间的地址，并将用新的MetaData来记录这个新指向。而且更灵活的是，还可以为这个Clone的虚拟卷再次创建快照，甚至CDP。

提示： 16.2.5节中会对这种可写以及创建二级快照或CDP的详细底层机制进行论述。

某个物理卷某时刻所生成的一份可写的Snapshot，就叫做这个卷某时刻的一份Clone。然而，这份Clone其内容没有被更改的部分是与源卷共享的，只有被更改之后的数据才是实实在在地存在于另一处的，所以，源卷没了，则Clone也就没了。这种Clone叫做虚拟Clone，虚拟Clone的好处是本身不占用空间，只有被更改的数据部分占用对应的空间。如果想对这个Clone做保留，保留其实体的数据，则需要将这份Clone与源卷所共享的那部分数据复制出来，加上后来被更改的数据部分（当然也可以不去更改这份Clone的内容），生成一个独立的卷。此时，这份Clone就会与源卷脱离关系，相当于依附于源卷的快照被切割开，或者说Split，这个过程叫做Split Clone。Split完成之后，就会生成一个实实在在的真正的物理上独立于源卷的卷，而且源卷对应的这个快照也就消失了，因为此时快照已经变成了实际的物理数据，也就是实

Clone，实Clone需要占用与源卷等同的物理空间。

2. 虚克隆和实克隆的比较

实克隆与虚克隆相比有两个好处。第一个是安全性方面。虚克隆是依托在物理源卷上的一个空中楼阁，一旦源卷发生任何物理上的损坏，比如承载源卷的底层物理磁盘发生损坏，例如，某Raid 5阵列中损坏多余两块磁盘之后，此时源卷物理上土崩瓦解，则其对应的Snapshot和克隆也就荡然无存了。而实克隆则是源卷某快照的一份真实物理复制，如果将克隆卷存放在与源卷不同的Raid组中，则即便源卷Raid组发生故障，那么克隆卷依然存在，可以直接将克隆卷挂载到主机继续使用。第二个是性能方面。虚克隆由于在物理上是与源卷共享大部分数据块的，如果对虚克隆进行读IO，则会与源卷的读IO发生争抢效应，从而源卷和虚克隆卷的性能都受到影响。而实克隆由于是存在于另一个Raid组，对实克隆的读IO由于是访问不同的物理磁盘，所以不会影响到源卷。这里需要注意一个问题，如果把实克隆卷放到与源卷相同的Raid组中，那么在读实克隆卷的时候造成的影响与虚克隆相同，因为它们共同争抢同一份物理Raid组。

3. 卷Clone的作用

卷Clone的一个最大的作用是可以瞬间生成针对某个卷的可写的镜像，而不管这个卷的数据量有多大。比如某企业有一个数据量2TB的数据库系统，为企业核心生产数据库的存储空间，非常重要。近期企业决定部署一套应用系统，但是需要利用现有的生产数据库进行上线前测试，IT管理员不得不将备份过的数据库恢复到一个和源库同样大的存储空间内，但是很尴尬地发现目前存储系统剩余的空间已经不足以满足要

求，不得不将现有存储系统扩容。这又要牵涉到一个大工程，而且测试完毕，新购买的空间又要被浪费，实在划不来。幸好现有存储系统支持卷Clone，IT管理员针对生产数据库所在的卷创建了一份虚拟Clone，并将其挂载到应用测试机上，应用程序成功启动，进入测试周期，幸好应用写入的数据很少，以至于Clone卷没有占用太多的存储空间，不需要额外购买存储空间了。一周后，应用测试管理员出现失误，将Clone卷中的大量重要数据删掉了，急急忙忙找到了IT管理员。IT管理员不慌不忙，从容地将这个Clone卷删掉，然后直接在生产数据库的源卷上又创建了一份Clone，映射给应用测试机，测试机挂载后，一切如故。

克隆可以用来研究平行宇宙、蝴蝶效应等理论。将某个时间点的影像瞬间复制成多份平行复制，然后让其并行地继续向下发展，调查后续各份平行数据的发展路径从而调查周围环境或者未发现的其他隐含因素对事件发展的影响。最简单和直观的例子：有100台硬件配置完全相同的PC，用同一张光盘灌入操作系统，同样的使用步骤和环境，但是随着时间的变化，每台PC上操作系统的行为开始产生差异，最后迥异。那么你怎么去调查到底是什么因素引起了如此巨大的差异？你可以编写一套数据监控统计挖掘分析报告的系统，针对同一份数据生成多份并行克隆，然后在其上进行模拟分析。甚至可以在虚拟现实系统中，对虚拟世界生成并行的多个克隆，然后调查各种行为对虚拟世界后续发展路径的影响。

4. 使用Clone卷来进行Rollback操作

1) 用虚Clone卷来Rollback源卷

如果某时刻，源卷中的数据发生逻辑上的错误，比如大规模病毒感染破坏、分区误删除、文件误删除等，此时可以选择直接使用虚Clone

卷进行Rollback操作，直接将原先被CoFW出去的数据覆盖回源卷。当然，如果Clone卷已经被其他程序更改，则也可以选择保留这些更改。如果选择保留更改，那么系统会先将前几节中描述的RoFW映射表与快照地址映射表中对应的地址逐条检测，如果发现某个地址在两份表中都有映射条目，证明Clone卷中对应的地址被改写过，则快照映射表中对应的IO仓库中的CoFW数据过期，在Rollback的时候直接覆盖RoFW表中的数据而无须先覆盖CoFW的数据再覆盖RoFW表中的数据，节约了资源。

2) 用实Clone卷来Rollback源卷

如果使用已经被Clone Split的实Clone卷来对源卷进行Rollback，则需要的步骤就会复杂一些。因为当初Clone被Split之后，系统会将源卷和克隆原来共用的数据块复制出来，如果在Split完成和Rollback之间的时段内，主机并未对源卷上的这些原本共享的数据块做太多更改的话，那么在Rollback的时候如果依然将所有原本共享的数据块也一同覆盖到源卷，则这样做就属于浪费资源了。

如何解决这个问题呢？此时快照指针已经不复存在，不可能再用快照恢复源卷了。要解决这个问题，就需要对这些共享的数据块在Split之后所发生的更改动作做追踪。这里人们想出一个办法，具体如下。

- 在Clone Split完成之后，立即为源卷和实克隆卷分别创建两份Bitmap，针对源卷的Bitmap中为全0，而实克隆卷的Bitmap中，对应之前被CoFW出的所有地址的位为1，其他全为0，为0的位则代表与源卷共享的数据块。
- Bitmap生成之后，系统才允许继续对源卷或者实Clone卷进行读写IO操作。

- 随后，当系统接收到任何针对源卷的写IO后，系统就会在刚才生成的全0源卷Bitmap中将对应IO地址的位置为1，并将IO写入源卷。同理，当接收到任何针对实Clone卷的写IO，也在刚才的那份针对实Clone卷的Bitmap中将对应IO地址的位置为1，如果已经为1则不做动作。

设想一下，经过这样的设计，源卷和实Clone卷在一段时间的数据更改之后，只有在两份Bitmap中都为0的位，才表示自上一次Clone Split之后两端均未发生更改的地址。而针对实Clone卷的初始Bitmap中只有当时与源卷共享的数据块才为0，这样就可以推导出，两份Bitmap中都为0的地址，其对应的数据块内容是完全一样的，所以在进行Rollback时，可以跳过这些数据块。

如图16-25所示，我们将这两份Bitmap做OR操作，得到的新Bitmap中，所有为1的地址，就代表需要从实Clone卷中将对应地址的数据覆盖回源卷，为0则跳过。覆盖成功之后，将当前处理的位重置为0，然后接着处理下一位，顺序扫描并处理结果Bitmap中的每一位。这样的设计可以大大降低Rollback过程中需要复制的数据，缩短Rollback时间。

图16-25 SplitClone和RollBack

3) 用源卷来Catchup实Clone卷到当前时间点

在Clone Split完成之后的某时刻，如果用户对上一个实克隆卷做了很多更改而且发现有逻辑上的问题，想对当前的源卷再次生成一份实克隆卷，当然，用户可以再次用传统的步骤，即先生成快照，然后Split进行数据复制，成功之后，删除原来的旧实克隆卷。但是这样做需要花费很长的数据复制时间，是否有一种办法在上一个旧克隆卷的基础上，将

源卷自上次Clone Split之后所发生的所有数据更改同步到上一次生成的实克隆卷上呢？

答案是肯定的。很显然，根据上一步中的OR操作之后的Bitmap，只有为0的位才是Rollback时源卷和实Clone卷上内容相同的数据地址，那么此时我们不是要从Clone卷复制数据覆盖到源卷，而是要做相反方向的动作，那么直接就可以根据结果Bitmap，将所有为1的位对应的源卷上的地址的数据复制出来并覆盖到实Clone卷。覆盖成功之后，将当前处理的位置为0，然后接着处理下一位，顺序扫描并处理结果Bitmap中的每一位。之后，实Clone卷就与当前源卷的内容相同了。这期间的数据复制跳过了两个卷原本内容就相同的数据块，节约了很多时间。

4) Clone Split期间如何处理主机读写IO

Split执行开始之前，系统需要首先创建一个针对源卷的RoFW数据映射表，Split开始之后，所有针对源卷的写操作都被重定向写到表中，一直到Split结束，将源卷与快照中共享的数据块复制到实Clone卷中之后，系统慢慢在后台将RoFW表中的数据覆盖回源卷，过程与上一步介绍的相同。

5) Rollback期间如何处理主机读写IO

Rollback过程都可以在后台执行，即对于前端来讲，Rollback可以被通告为立即完成。在Rollback后台执行期间内，如果遇到主机端的读IO操作，则系统必须参考OR运算之后的结果Bitmap。如果IO的地址对应的位为1，则从实Clone卷中读取对应地址的数据返回给主机（所以在Rollback期间，实Clone卷是不允许写的，否则会不一致）；如果对应的位是0，则直接从源卷对应的地址读取数据并且返回给主机。如果遇到

写操作，依然首先查找结果Bitmap，对应地址的位如果为0，则直接将IO写入源卷；如果对应的位为1，则系统将IO写入源卷对应地址之后立即将结果Bitmap中对应的这一位置为0。

6) Catchup期间如何处理主机读写IO

总体来讲，这一步与上一步的操作步骤本质上相同，但是需要增加一步。由于Rollback时实Clone卷不允许接受写IO，那么同理，Catchup时源卷也就不能接受写IO。但是这样做是不允许的，除非是前台Catchup，但是这样会对主机应用造成暂停影响。解决的办法依然是使用RoFW大法了。在Catchup执行之前首先创建一个针对源卷的RoFW数据映射表，所有在Catchup期间针对源卷的写IO都被重定向到这个表中，读IO则首先查询这个表，如果找到对应的地址则从这个表而不是源卷中读取数据并返回给主机，如果是写IO则也先查询这个表，如果对应IO地址已经存在条目则覆盖，不存在则插入。当Catchup完成之后，系统再将RoFW表中的数据慢慢在后台同步回源卷。

卷Clone是依托在快照技术上的一个高附加值产物，其附加价值就体现在一个RoFW映射表上，从而使快照可写。实现这个技术不是什么难事，但却能得到高回报率。

5. EMC公司卷Clone产品简述

1) Snapview系列

Snapview是运行在EMC公司中低端存储系统Clariion CX平台的操作系统FLARE上的一个软件模块，它又被分为Snapview/snapshot和Snapview/clone两个功能选项。前者代表普通快照功能，后者则代表卷

Clone功能。

Snapview/snapshot在进行CoFW时会以64KB为单位进行CoFW，也就是说，快照生成之后，如果有某个针对源卷的写IO仅为4KB，那么系统依然会将这4KB所在的64KB单元全部复制到IO仓库（EMC的说法是Reserved LUN）中存放。如果随后某个写IO再次落在了这64KB单元地址的任何地址段，系统也不会再进行CoFW操作。这种提高CoFW单元粒度的方法一方面降低了维护更细粒度地址映射表的开销，另一方面也显著降低了CoFW对系统写性能的影响所持续的时间。唯一一个坏处就是会浪费IO仓库中一定的空间，可以说是以空间换性能。

Snapview/Clone使用一种叫做Copy On First Access（CoFA）的机制，当挂载并使用克隆的那台主机发起针对这个克隆卷的任何IO时，不管是读还是写，如果IO的目标地址块是首次被读或者写，那么系统将从源卷中将对应的块（或者更大的粒度，见上文）复制出来放到克隆卷空间里存放。同时，针对源卷的首次覆盖写入时当然也会触发复制，但是针对源卷的块的读操作是不会触发复制的，只有针对克隆卷的首次读才会触发复制。这样做的目的是为了降低克隆卷对源卷的性能影响，但是却增加了存储空间的需求量。

如图16-26所示为EMC Clariion配置界面，其中Fracture、Synchronize和Reverse Synchronize分别对应上文中的 Clone Split、Catchup和Rollback。

图16-26 Fracture

2) Timefinder系列

Timefinder是运行在EMC公司高端存储系统Symmetrix DMX平台的

操作系统Enginuity上的一个软件套件模块。它包括TimeFinder/Snap、TimeFinder/Clone、TimeFinder/Mirror、TimeFinder/CG（Consistency Group）等几个组件。

TimeFinder/Snap、TimeFinder/Clone分别对应了快照和卷Clone的功能。TimeFinder/CG则实现了一致性组功能，关于一致性组技术会在后面的章节详细地论述。

这里需要着重提一下TimeFinder/Mirror功能。传统的Clone卷是基于对源卷在某一时间点的快照而作成的，但是TimeFinder/Mirror则不依托快照来实现，而是直接对源卷进行实时的同步镜像，针对源卷的任何写IO都被实时同步到一个镜像卷中。

某时刻，用户可以发起Split动作，使这个镜像卷脱离源卷而成为一个独立的卷，也就相当于在这个时刻创建了一份卷Clone，只不过这份卷Clone在Split之后立即就是一个物理独立卷了，而不需要任何数据复制过程。Split Mirror之后，源卷与切开后的镜像卷也可以各自记录一份Bitmap，就像上文中所说的。而随后，可以选择利用源卷来Catchup这个Clone卷，或者选择使用Clone卷来Rollback源卷。EMC将这种镜像卷称为BCV，即Business Continuous Volume。其作用与传统Clone卷相同。唯一优点就是Split之后立即可用，无须等待数据复制。最大的缺点则是由于数据完全与源卷同步，一旦源卷数据发生逻辑错误，而发生错误之前又没有手动Split，则镜像卷的数据也同样变为逻辑不一致的了。

16.2.5 Continuous Data Protect（CDP，连续数据保护）

SNIA对于CDP给出了一个定义。CDP（持续数据保护）是一种在不影响主要数据运行的前提下，可以实现持续捕捉或跟踪目标数据所发

生的任何改变，并且能够恢复到此前任意时间点的方法。CDP系统能够提供块级、文件级和应用级的备份。

有一类所谓的Near CDP产品，可以生成高频率的快照，比如一小时几十次、上百次等。用这种方法来保证数据恢复的粒度足够细。

CDP是这样一种机制，即它可以保护从某时刻开始卷或者文件在此后任意时刻的数据状态，也就是数据的每次改变，都会被记录下来，无一遗漏。这个机制乍一看非常神奇，其实它的底层只不过是比快照多了一些考虑而已，下面我们就来分析它的实现原理。

1. 应用级和文件级的CDP

所谓应用级CDP，是说对数据的连续保护机制是发生在应用程序层的，换句话说，由应用程序自己对自己的数据加以连续保护，记录和保存每一笔更改。应用级CDP的典型例子就是比如Oracle和DB2等各种数据库系统，数据库系统对每一笔交易都会进行日志记录，在归档日志模式下，所有曾经对数据库进行的更改操作均会被打入时间戳并记录到日志中，老日志不断地被归档存放以便为新日志腾出空间。当数据库发生问题的时候，利用归档的日志，就能够将数据库状态恢复到任何一个指定的时间点，数据库会顺序读出库中的每一笔交易然后将其重放

（Replay），对应的数据重新写入数据库文件。重放完成后，还需要进行Redo和Undo操作，即检查日志中最后一个CheckPoint一致点处，一致点之后发生的交易全部回退。回退完成后，数据库便处于一个一致的状态并且可用。应用层CDP不需要任何其他程序的辅助，不需要任何特殊的存储系统功能，完全由应用程序自身就可以完成。应用级CDP是最纯粹、最厚道、最彻底、最实用的CDP。

文件级CDP就是通过监视文件系统动作，文件的每一次变化（包括实际数据或者元数据的变化，比如重命名、删除、裁剪等属性的改变）以日志的形式被记录下来。CDP引擎分析应用对文件系统的IO数据流，然后计算出文件变化的部分，将其保存在CDP仓库设备（存放CDP数据的介质）中，可以针对每个文件生成单独的日志链。可以对一个文件，或者一个目录，甚至一个卷来监控。文件级的CDP方案，一般需要在生产主机上安装代理，用来监控文件系统IO，并将变化的数据信息传送到CDP仓库介质中，或者使用本地文件系统或者磁盘的某块额外空间来充当日志仓库。文件级的CDP，能够保证数据的一致性。因为它是作用于文件系统层次，捕获的是完整事务操作。所有的文件版本管理软件都可以算作是文件级CDP的实现。

其实日志型文件系统自身也可以算作是一个粗线条的CDP实现，因为日志型文件系统自身也会记录每一笔操作记录和数据，只不过其日志是循环的，并非归档模式，同时默认的日志方式是只记录元数据更改而不记录实际数据，并且也不提供用户自定义回溯时间点的功能。如果能够直接在文件系统模块中或者外嵌一个模块来针对每个文件记录归档模式的元数据+实际数据日志，那么恢复的时候就可以指定某个文件的某个时间点进行数据回滚了。

2. 块级的CDP

块级的CDP，与应用级和文件级CDP实现思想相同，其实就是捕获底层卷的写IO变化，并将每次变化的块数据打入时间戳并且保存下来。这里先不探讨具体产品的架构，而只对其代码层设计原理做一个细致的描述。后文中则会对块级别CDP整体架构做更加细致的论述。

提示：

以下对于CDP的描述引自敖青云的博客

(aoqingy.spaces.live.com)。

CDP起源于Linux下的CDP模块。它持续地捕获所有I/O请求，并且将这些请求打上时间戳标志。它将数据变化以及时间戳保存下来，以便恢复到过去的任意时刻。

在Linux的CDP实现中，包含下列三个设备。

- 主机磁盘设备 (host disk)
- CDP仓库设备 (repository)
- CDP元数据设备 (metadata)

CDP代码对主机磁盘设备在任意时刻所做的写操作都会记录下来，将实体数据按顺序写入CDP仓库设备中，对于这些实体数据块的描述信息，则被写入到CDP元数据设备的对应扇区。

元数据包含以下信息：

```
struct metadata {  
    int hrs, min, sec; 该数据块被写入主机磁盘设备的时间  
    unsigned int bsize; 该数据块以字节为单位的长度  
    sector_t CDP_sector; CDP仓库设备中对应数据块的起始扇区编号  
    sector_t host_sector; 该数据块在主机磁盘设备中的起始扇区编号  
};
```

图16-27反映了主机磁盘设备和CDP仓库设备之间的关系。CDP仓库设备中按时间顺序保存了对主机磁盘设备的数据修改。A为主机磁盘设备上的一个扇区，该扇区在9:00和9:05分别进行了修改，它在CDP仓库设备中对应的扇区分别为A1和A2。

图16-27 CDP仓库与主机磁盘设备间的关系

图16-28反映了CDP仓库设备和CDP元数据设备之间的关系，它们的写入顺序一一对应。CDP仓库设备中的一个元数据，对应CDP元数据设备中一个I/O请求，实际上可能是多个扇区。具体扇区数由元数据中的bysize指定，而起始扇区位置由CDP_sector指定。

图16-28 CDP仓库设备与CDP元数据设备间的关系

全局变量maddr保存了下一个I/O请求，在CDP仓库设备上执行的地址（起始扇区编号）。maddr的初值被定义为宏START_METADATA（0）。

```
unsigned int maddr = START_METADATA;
```

当一个写请求到来时，对应数据被写到CDP仓库设备中，这时所做的操作如下：

- （1）将写入CDP仓库设备的数据块起始扇区编号设置为maddr；
- （2）根据要写入主机磁盘设备的数据块的扇区数目增加maddr。

这时，用户要将这里写入CDP仓库设备的数据块编号记录下来，以便构造对应的元数据。

全局变量taddr保存了下一个I/O请求，对应的元数据，在CDP元数据设备中保存的地址（起始扇区编号）。taddr的初值被定义为宏START_METADATA（0）。

```
unsigned int taddr = START_METADATA;
```

当一个写请求到来时，对应的元数据被记录在CDP元数据设备中。

为了简单起见，在元数据设备上，一个扇区（512B）只保存一个元数据信息（只有32B），这样浪费了大量的存储空间，但对元数据设备的处理却非常简单：

（1）将写入CDP元数据设备的元数据起始扇区编号设置为taddr，长度为1个扇区；

（2）将taddr增加1；

（3）请求处理过程。

请求处理过程是从make_request函数开始的。考虑到读请求处理的相似性，甚至更为简单，这里只分析对写请求的处理过程。首先获得当前的系统时间，然后写请求bio结构（为便于说明，这里记为B）被分为三个写请求bio结构（分别为B0、B1和B2），如图16-29所示。这三个bio结构的作用如下。

图16-29 三个写请求bio结构

- B0：将数据块写到主机磁盘设备。
- B1：将数据块写到CDP仓库设备。
- B2：将元数据写到CDP元数据设备。

同其他块设备驱动程序的实现一样，从B复制产生B0、B1和B2，然后重定向它们要处理的设备，即bi_bdev域。另外一个大的变动是重新设置了bi_end_io域，用于在I/O请求完成之后进行善后处理。

为了处理善后，还要将B0、B1和B2的bi_private指向同一个

CDP_bio1结构。从这个结构能够回到对B的处理。

```
struct CDP_bio {  
    struct bio *master_bio; 原来的bio，通过这个域用户可以从B0、B1、B2找  
    struct bio *bios[3]; 如果IO为WRITE，这个指针数组分别指向B0、B1、B2，  
    需要这个域？  
    atomic_t remaining; 这是一个计数器，在后面会解释  
    unsigned long state; 在I/O完成方法中使用  
};
```

善后工作的主要目的是在B0、B1和B2都执行完成后，回去执行B，为此需要一个“have we finished”计数器，这就是原子整型变量remaining。在构造B0、B1、B2时分别递增，同时在B0、B1和B2的I/O完成方法中递减，最后根据该值是否递减到0，来判断B0、B1和B2是否都已经执行完毕。为了防止B0在构造后、在B1和B2构造之前就执行到B0的I/O完成方法，从而使得remaining变成0这种错误情况，我们没有将remaining的初值设置为0，而是设为1，并在B0、B1、B2都构造完成执行递减一次。

B0、B1、B2都执行完成之后，进行如下的处理。

- (1) 调用B的善后处理函数。
- (2) 释放期间分配的数据结构。
- (3) 向上层buffer cache返回成功/错误码。

提示： 对B2的构造，这个bio结构需要处理的是元数据。时间戳已经在进入make_request时获得了保存，而对主机磁盘设备

操作的起始扇区和长度从B中可以获得，对应的CDP仓库和CDP元数据的起始地址分别保存在全局变量maddr和taddr中。

数据恢复过程

用户可以将数据恢复到以前的任意时刻。CDP实现代码中提供了一个blk_ioctl函数，用户空间以GET_TIME为参数调用该函数，将主机磁盘设备中的数据恢复到指定的时间点。恢复的过程分为以下几个步骤。

（1）顺序读取CDP元数据设备的所有扇区，构造一个从主机磁盘设备数据块到CDP仓库设备的（在这个时间点之前）更新数据块的映射。其结果保存在以mt_home为首的（映射表）链表中，如图16-30所示。

图16-30 CDP恢复过程

这里需要构造taddr个对CDP元数据设备的读请求，每个请求读取一个扇区。在这些请求的I/O完成方法中，从读到的数据中构造元数据，并递减计数器count。

如果元数据中的时间戳早于或等于指定的恢复时间点，则需要添加或修改mt_home链表的元数据结构。

提示： 这些项是以host_sector为关键字索引的，因此添加或修改取决于前面是否出现对同一个host_sector的修改。在以顺序方式读取的过程中，可以保证host_sector（在指定的恢复时间点之前）的最新修改CDP_sector会出现在这个链表中。

由于计数器count为taddr，如果它递减为0，说明CDP元数据设备中

的所有数据均已读出并处理，这时就可以继续往后面执行。

(2) 从CDP仓库设备中读取这些更新的数据块，构造以mt_bi_home为首的链表。

同上面的处理类似，我们需要为mt_home链表中的每一项构造对CDP仓库设备的读请求，每个请求在CDP仓库设备的起始编号取决于CDP_sector域，长度则根据bysize而定。这个请求读出的数据需要被写入到主机磁盘设备中，为此在读请求I/O完成函数中，构造一个对应的往主机磁盘设备的写请求bio，该写请求的起始编号取决于host_sector域，长度根据bysize而定，而要写入的数据是刚刚从CDP仓库设备中读出的数据。另外，在读请求I/O完成函数中，还要递减一个计数器，当该计数器递减到0时，说明用户已经全部处理了mt_home链表中的项，这时会得到一个以mr_bio_home为首，每项中都指向一个bio结构的链表。

```
struct list_head mt_home; //BIO更新链表
struct most_recent_blocks { //BIO更新表项
    struct bio *mrbio;
    struct list_head list;
};
```

(3) 将mt_bi_home链表的数据块都恢复到主机磁盘设备中。

这个操作相对比较简单，用户只需要在主机磁盘设备上执行mt_bi_home链表的每一个bio请求项即可。当然还要在这些请求项的I/O完成方法中做善后处理，即如果所有请求项都已经执行完毕，则释放mt_home链表和mt_bi_home链表。

3. CDP架构模型演绎

前面主要给出了CDP底层的设计思想和思路，本节将对CDP的具体实现做一个架构模型猜想和分析，在下一节中则介绍几个目前市场上厂家的CDP产品，通过分析其产品架构从而对CDP进行更加深刻的理解。

演绎之前，我们再来总结一下CDP的基本原理：对需要保护的数据进行连续的监视，每当有写IO发生，视设计方法不同，将这份IO的数据打上时间戳，保存到另外的空间中（RoW）或者将被覆盖的数据先复制出来打上时间戳并保存到另外的空间中然后再覆盖新写入的数据（CoW）。这里为何没有用CoFW或者RoFW呢？因为我们这里做的不仅仅是一次Snapshot，而是连续数据保护，记录每个IO，所以怎么会允许第二次覆盖写就不Copy或重定向了呢？每次写都要Copy或者Redirect，除非用于保存变化IO的仓库已满或者已经达到用户所设定的保护期限阈值。

说明： CDP思想的核心可以说就是时间戳，任何IO都必须有
时间戳，CDP引擎在遍历整个链的时候，除了要参考FS或者
VM自身的元数据映射链或者映射图表之外，还要参考时间戳
链，能够做到对于每一个给出的时间点，CDP引擎都可以从源
数据区和变化IO仓库区从链表/映射图以及时间戳链中遍历抓
取出一份完整的对应所给出的时刻的源数据卷的元数据链，从
而得到了完整的实际数据卷的一份影像（image）。这份影像
就像Snapshot一样可以被只读挂载以用于恢复数据或者备份到
磁带，或者像卷Clone一样可写。如图16-31所示为作者自演绎
的一种CDP产品模型。

图16-31 自己演绎的CDP模型

图16-31所示的是一个假想中的CDP实现架构。特别需要注意的是

图中所有箭头的位置和指向，每个位置和指向都是有特殊意义的，方向向上的箭头代表读IO操作，向下则代表写IO操作。下面就这个架构模型来做一个论述。

一台主机后端通过FC交换机连接了一台磁盘阵列，磁盘阵列映射了若干LUN给这台主机使用，并且主机还有本地磁盘，受保护的文件和卷既有一部分存放在磁盘阵列中，又有一部分存放在本地磁盘中。其前端通过以太网交换机连接了一台CDP服务器。主机上安装有CDP客户端套件，包括的主要模块有：CDP主代理模块、块IO捕获虚拟设备模块和文件IO过滤驱动模块。

1) 文件IO过滤模块

这个模块作用于文件系统层旁路以及卷管理层之上，其作用是监视每个受到CDP保护目录下的文件的所有写操作，一旦发现某个受保护的文件有写入操作，则将这个操作所对应的数据和偏移量一同通过以太网发送到CDP服务器端（RoW方式下），或者先将受保护的文件对应的即将被覆盖的部分的现有内容读出然后附带对应的偏移量信息一并发送到CDP服务器端后再将待写入的数据写到对应的磁盘空间中（CoW方式下）。文件IO过滤层是实现文件级CDP的一个必需组件。而实现卷级或者Block块级的CDP，就需要在卷管理层之下的Block IO链条之间插入一层，也就是图示的块IO捕获虚拟设备层。

2) IO捕获虚拟设备层

这个组件在卷管理层之下生成了一个虚拟设备，卷管理层直接对这个设备进行IO，从而将任何试图写入受CDP保护的卷的IO操作截获，与文件IO过滤组件作用方式相同，根据不同设计方式比如CoW或者

RoW，将对应的数据和LBA地址段信息发送到CDP服务器端。这两个组件并不是所有IO都监控，它们只会向操作系统API注册受CDP保护的目录或者卷的监控过滤权限，上层对未受保护对象的写IO操作均会Bypass掉这两个层直接写向原有的下层。图中的箭头明确标示了各个层次之间的关系。

3) CDP主代理模块

这个模块是受保护的客户端主模块，负责协调与CDP服务器的通信和数据传输，以及适配上述的两个模块。如果说上面两个模块是两杆枪，则客户端主代理模块就是前线指挥者，CDP服务器就是后方总指挥外加战俘集中营。主代理模块还负责把每个从上面两个模块接收到的写IO数据打上时间戳并发送给CDP服务端模块保存。

如果使用RoW方式，则上层的每个读IO也均会经过文件IO过滤驱动层，过滤驱动将IO的地址传送给主代理模块，主代理模块向CDP服务器发出查询请求（良好的设计情况下无须每次都查询，代理应当定时从CDP服务器端将地址统计表拉过来做缓存），如果这个IO的地址恰好是以前曾经被RoW过的地址，则CDP服务器从IO仓库中将这段地址对应的实际数据提取并发送给客户端主代理，代理将收到的数据传送给文件过滤驱动，然后文件过滤驱动再将数据传送给上层请求者。可以看出，RoW方式的流程过多，效率非常低下，但是可以利用另外一种方式来既用到RoW不消耗额外的IO资源这个优势，又能屏蔽掉读操作对主机的影响，这种设计思路将在下文描述。

在利用CDP做实恢复（对应虚拟影像来讲）的时候，主代理模块还负责从CDP服务器提取对应时间点之前的所有被保存的IO数据并将恢复的数据覆盖到对应的目录或者卷。在利用CDP做虚拟影像挂载读写的时候

候，上层针对虚拟卷或者目录的每次读或者写IO，都需要代理根据本地所缓存的地址统计表来判断对应IO的实际数据在源目录/卷还是位于CDP服务端的IO仓库中，并去将数据取回并返回给请求者。

4) CDP服务器端

CDP服务器端是整个CDP系统的总控制台和IO仓库。服务端负责监控和记录所有受保护的客户端主机上的代理程序状态以及目录/卷的状态、传输状态等。服务端负责响应代理端的查询请求并在查询命中时将保存在仓库中的IO数据返回给代理端。服务端还负责接收各个代理端传送过来的需要保存的IO，并将这些IO数据加以分类并存放到本地的存储空间中，并做好时间错索引/地址段索引等重要步骤以便提高查询速度。服务端还负责提供用户配置接口，用CLI/GUI方式接收用户的配置，比如保存期限、空间配额、备份策略、传输模式、带宽控制、用户权限控制等。

CDP服务端可以是一台普通的服务器，安装对应的软件，也可以是内嵌软件的成品一体化硬件服务器。至于CDP服务器的存储空间，则需要根据受保护的总容量、处理能力和具体需求等来配置，服务器机箱内本地磁盘以及通过FC或者SAS之类的适配卡连接的外置扩展柜或者第三方独立存储系统，均可作为IO仓库存储空间。

5) 源卷数据被保护的详细步骤阐述

有一台主机，其上有一个100GB大小的卷或者分区，需要对其进行CDP连续保护，保护期限为一周，也就是说只保留一周前到现在的所有变化数据即可。CDP服务器已经安装配置完毕。

(1) 首先在这台主机上安装CDP客户端代理模块，安装完毕后需

要重新启动。重启后，对这个卷的IO更改率进行评估，得出结论，IO仓库容量为源卷容量的20%即可。

(2) 打开代理程序主界面，在其中申请20GB大小的IO仓库，并将其分配给待保护的卷。配置确定后，将其保存，此时代理与CDP服务器进行通信，服务器端自动在其本地存储仓库内分配了一块20GB大小的空间并记录这个空间的使用者和卷的信息，并且对这个空间做一些元数据的初始化操作，等待接收客户端发送过来的IO数据。

(3) 服务端完成配置后，向客户端代理发送成功信号。客户端代理在GUI界面中将成功的状态显示出来并等待管理员的下一步指示。此时，数据保护还没有启动，需要管理员手动启动。管理员手动将数据保护引擎启动，此时，块IO捕获引擎开始工作，将需要被保护的IO数据源源不断地发送给代理主模块，主模块打上时间戳后将IO数据传送到服务端保存。

(4) 服务端收到对应客户端代理的IO数据后将其存放到对应的IO仓库中，按照设计的保存格式将其写入硬盘，并做好MetaData的更新。目前几乎所有的CDP设计都类似，即为每个Block地址形成一条单独的数据块链条，每个针对源卷这个地址的写IO所产生的CoW或者RoW出来的块都会被追加到这个链条末尾并打入时间戳。

(5) 随着客户端的IO数据不断传来，IO仓库将变得越来越满，在仓库未全满之前，比如一周之后，服务端引擎开始根据时间戳来判断并删除时间戳早于一周的所有IO数据以便清空部分空间。

(6) 当服务端的IO仓库内容量达到所设定的阈值的时候，将会产生告警信息提示管理员将IO仓库扩容。当满仓之后，针对这个卷的CDP

引擎停止在当前的时间点（RoW模式下不可全停，因为随时要响应上层的读IO请求），等待管理员进一步指示。当管理员删除一些早期时间戳IO数据之后，或者扩容IO仓库之后，重新启动CDP引擎，停止时间点与启动时间点之间客户端卷变化的数据没有被记录，所以这段时间是空白期，数据不能回溯到这期间的时间点。而且，如果使用RoW方式的话，一旦IO仓库满仓，则不但不能回溯到空白期，而且还会发生逻辑上的矛盾。

比如，满仓之后，假设源卷有一个写IO是针对LBA1024的，并且LBA1024在满仓之前就已经被覆盖过多次，所以这个地址的多份IO数据被冠以不同的时间戳保存在仓库中，此时，客户端代理由于仓库满仓而停止了IO捕获，那么这个IO应该如何处理？难道要直接写到源卷将原来的数据覆盖么？如果这样做，随后上层发起一个读IO请求，读取LBA1024的话，则客户端代理经过查询服务端后，会将服务端仓库中对应这个地址的最后一个时间戳的数据返回，而忘记了这个地址的最后一次写动作其实是直接写到了源卷了，并没有RoW，此时如果果真返回了仓库中的数据而不是源卷上这个地址的数据，则会对上层应用造成重大影响，轻则数据错乱，重则崩溃，这是绝对不容许发生的事情。

要解决也不难，客户端代理在让这个写IO直接写到源卷之前，先做一下记录，这份记录包含所有在IO捕获引擎停止之后发生的，并且服务端仓库中包含对应地址的数据的写IO信息，这样，上层写完之后再发起读这个地址的请求的时候，客户端代理程序优先查询这份记录，匹配则直接将IO定向到源卷。当服务端IO仓库有足够剩余空间之后，CDP引擎启动，继续接受RoW的IO数据，一旦有某个IO是针对这份新记录中的某一条地址的，则客户端代理将IO打入时间戳并发送到服务端仓库之后，立即将这条地址从记录中删除，以后上层针对这个地址的读IO便会

再次被重定向到服务端仓库里对应这个地址的最后一个时间戳的数据。

提示： RoW方式流程确实够复杂够变态，所以，本模型宣告抛弃这种方式，全面转向CoW方式，大家可以自己演绎一下，CoW方式并不会出现上述的问题，这里就不再具体论述了。使用RoW模式的情景会在另一个模型中介绍，而且这也是目前主流产品都在使用的模型。

6) 数据恢复/回溯时的详细步骤阐述（基于**CoW**方式）

（1）周一早晨，管理员对这个卷做了一次Snapshot。周二上午，一位员工小糊找到管理员说他在周一下班的时候，在编辑完这个卷上保存的一份重要文档之后，竟然稀里糊涂地将其误删除了，问能不能恢复回来。管理员说，恢复太困难了，得耽误很长时间，而且很麻烦。小糊中午请管理员吃了顿大餐。

（2）下午上班，管理员首先想到了用Snapshot做恢复，但是不巧，最后一份Snapshot是周一早晨做的，而那老兄在周一下午对它进行了编辑，所以这个Snapshot并不包含这份被编辑后的文档。幸好，这个卷正受到CDP的保护，就让时光来倒流吧。打电话询问了那位糊涂兄，得知编辑完之后大概是周一下午5点整左右，什么时候删除的，他也不知道，他只知道自己5点半下班走人的。这可难为管理员了，不知道准确时间的话，只能摸着石头过河，用二分法吧，看来中午这顿饭吃对了。

（3）管理员打开文件服务器上的CDP客户端代理配置界面，选择了Rewind Wizard，然后选中要时光倒流的卷，下一步，出现一个滑动条，管理员首先将滑块拖动到周一17点15分，也就是糊涂兄开始编辑到

下班走人这期间的中点。然后单击Mount。当鼠标单击Mount的一刹那，客户端代理迅速地将指令发送给了CDP服务器端，告知本代理需要提供服务，请准备一下相关的资源，缓存Buffer进程线程都给准备好，不得怠慢！服务端迅速准备好，然后通知代理：准备好了！代理收到服务端反馈后，立即在文件服务器上虚拟出一个新卷并且挂载到一个盘符下。

（4）管理员打开“我的电脑”，出现了一个新磁盘F盘，总容量与受保护的卷一样，一看就知道是源卷在周一17点15分时刻的虚拟影像了，毫不犹豫地双击进去。

（5）客户端代理没想到管理员手竟然这么快，刚挂载上就要双击，本来想歇一阵子看来不行了，来活了！谁呀，一看是文件系统老大，这可不敢怠慢，主机上的红人啊！“文兄，有何指示？”代理点头哈腰地说。“嗯，老大有指示，让我来视察一下F区大堂都有些什么人儿啊，我查了查地图，F区大堂在LBA1024到LBA2048这段地址上，你给我跑趟腿，去看看然后告诉我结果。”代理赶忙先问了问CDP服务器端仓库中对应LBA1024到LBA2048这些数据块的各自链条中有没有带周一17点15分的戳子的，有的话全部给发过来。得到CDP服务端的答复，LBA1024到LBA1096有，内容也发过来了，但是LBA1097到LBA2048，在对应的链条中没找到带周一17点15分戳子的。代理一下明白了，既然对应这个戳子的数据不在CDP服务端仓库里，那一定就在原处了，赶紧去了趟源卷，把这段地址的数据也拿了过来。然后点头哈腰地给文件系统过目，文件系统拿到数据后，把数据给了管理员老大。

（6）（上述情节发生在半秒钟前）管理员双击进去之后，看到其中有个目录叫做“难得糊涂”，想都没想，老糊的那个文档一定就在这里面了，于是又双击进去。

(7) 文件系统喘着粗气又来到代理这里，上气不接下气地说：“竟然又让老子跑腿，他就不会一次到位么！哎我说，你再跑个腿，老大让我再看看F区下面的难得糊涂楼里面都有些什么人儿，我查了查，在地址LBA8192到LBA10240这段上。”代理心想，你这不也在让我跑腿么，扯淡。和上次一样，把对应的数据都给了文件系统。老文接过来一看：“空的？”代理扭捏道：“额.....我可看不懂这些内容。”老文一皱眉：“你小子，骗我的话有你好看！”

(8) (上述情节发生在半秒前) 管理员打开老糊的这个目录一看，什么都没有，顿时明白了：糊涂兄一定是在17点到17点15之前某个时间把文档给删了。太背了，还得再用一次。管理员打开代理程序配置界面，选中已经Mount的虚拟卷，单击Unmount将其卸载。之后又打开Rewind Wizard，滑动条拖到17点07分，单击Mount，然后“我的电脑”双击进入新挂载的虚拟卷，进入糊涂兄目录，一看，很好，这次文档出现了，证明糊涂兄一定是在17点07分到17点15分之间将其误删除的。但是07分的这份文档影像并不见得就是删除前最后的影像，还得继续找。定位到17点11分，挂载，进入目录，文档存在；定位到17点13分，进入，文档存在；定位到17点14分，进入，文档存在；17点14分30秒，存在；14分45秒，存在；14分52秒，存在；14分56秒，存在；14分58秒，存在；14分59秒，存在.....噢，小样儿，你挺能整人儿啊，在17点14分59秒和17点15分之间删除的，算你狠，给我出这等难题。看我的！再次打开滑动条，傻眼了，最小单位是秒，转眼一看，下面有个输入框，旁边还写着“请输入时间点，最小单位，毫秒”。我算算哈，1秒=1000毫秒，对吧，对。啊！！！二分1000，好么，谁怕谁.....

(9) 文件系统踉踉跄跄来找代理：“我说，老兄，大爷，帮帮我.....快，老大发疯了，快去给我查查.....。”

(10) (两小时后) 下午下班前, 管理员精神恍惚地来到糊涂兄面前: “你小子, 你知道我为了恢复你这份破文档, 费了多少劲么? 啊! ”。糊涂兄拿过文档一看, 一点数据也没丢, 乐呵呵说道: “小管, 辛苦了啊! 会有补偿地——”。管理员一愣, 只听背后Boss走过来: “呵呵, 小管啊, 做的不错啊, CDP用得很熟练了, 这份文档如果丢失, 将会带来惨痛损失啊, 做的不错, 这月奖金多给你50%! ”小管顿时眼泪哗哗.....

7) 模型改进演绎

细心的读者可能已经发现了, 上面的模型存在几个问题。一是, 源数据卷或者目录中没有被写覆盖的文件部分或者Block, 并未被传送到CDP服务端做备份, 而只是将被覆盖的数据 (CoW) 或者新数据 (RoW) 传送到CDP服务器。如果被保护的目录或者卷由于某种原因整体损坏, 或者整体被删除, 则CDP服务器上保存的IO数据就会变得无用。虽然在源卷或者源目录完好的情况下, 利用CDP可以回溯到之前的任何时间点的状态, 但是由于没有对源处的数据进行备份, 所以并没有达到严格的备份和保护数据的作用。第二个问题, CDP服务器与客户机之间只是通过以太网互联, 作为数据存储系统的IO传输通道未免有些捉襟见肘。

基于以上两个因素考虑, 我们把这个模型做一下改进, 改进后的模型拓扑图如图16-32所示。

图16-32 改进后的模型

上图中, 主机端逻辑模块没有变化, 但是主代理模块的功能有变, 原来的模型中, 主代理模块需要在恢复的过程中负责在本地主机虚拟一

个对应时间点的卷影像，并负责接收这个卷的IO并且通过以太网向CDP服务端引擎发起查询和数据传输操作。新模型中，这个工作全部被转移给了CDP服务端的虚拟卷映射引擎来完成，主机端代理模块主要负责状态监控和报告以及提供用户配置接口，控制信息的传输依然通过代理模块经过以太网来与服务端通信。同时，我们可以看到CDP服务器在新模型中也连接了FC交换机，目的是让捕获的IO通过FC网络而不是以太网来传输，加快速度。

运行在CDP服务端上的虚拟卷映射管理引擎的作用有两个。一是在数据持续保护期间，生成并通过FC通道或者以太网ISCSI通道向主机端映射一个与源卷相同容量的卷空间，IO捕获层首先将源卷上已经存在的数据全部同步到新映射的卷，完成之后，主机端针对源卷的每个上层IO便会都镜像一份，同时写入源卷和CDP服务端映射过来的卷。此时，这个块IO捕获层就充当了类似LVM的卷管理层。CDP服务端接收到每个写IO之后，打入时间戳，并将服务端的镜像卷中等待被覆盖的BBlock进行CoW操作，复制至IO仓库中保存。

这样算来，CDP服务端所需要的空间等于源卷空间再加上IO仓库占用的额外空间，而第一个模型则只需要IO仓库空间。虚拟卷映射引擎的第二个作用，便是在回溯恢复的过程中提供对应时间点的源卷的虚拟影像卷，也就是提供第一个模型中客户端代理在恢复过程中所执行的功能。有一点不同的是，第一个模型中的卷是完全在本地虚拟，而第二个模型中的卷则是在CDP服务端虚拟的，通过FC通道或者以太网ISCSI通道映射给主机；其次，虚拟卷映射引擎向IO仓库管理引擎发出查询等通信的过程均无须经过任何外部网络，直接在内存中进行，与第一个模型相比，速度和效率相对提升。

上述均以卷级CDP为基础，对于文件级的IO过滤捕获，还是通过以

太网来传递比较好控制，如果非要将文件级IO数据也通过FC网络进行传输，不是不可以，而是开发起来相对复杂，毕竟基于以太网的TCPIP是比较开放和成熟的接口。而SCSI Block级别的捕获由于FC协议栈已经提供了完善的FCP层支持，所以开发起来相对方便。

经过改进之后的模型，优点是达到了对数据全部保护万无一失的要求，并且提升了数据传输速度，而且降低了主机资源的耗费。不足则是CDP服务端需要比源卷更大的存储空间。这种架构还能够实现一个非常实用的功能，就是裸机恢复。比如，用CDP来保护主机操作系统所在的磁盘或者分区，一旦某时刻，主机操作系统崩溃，或者中毒已深，或者一些重要配置比如Domain配置给倒腾废了，如果没有CDP保护，重装个系统加上应用程序安装配置怎么也得半天时间；有了CDP保护，就大不相同了，此时在CDP服务端上将系统源卷在崩溃前某时间点的一份可写的影像通过FC通道Map给这台主机的FC适配卡的WWPN地址，然后主机在BIOS中设置通过FC卡启动，完毕后重启，即可从CDP服务端的这份虚拟影像卷来启动操作系统，启动后的状态与崩溃之前完全一致。

操作系统启动之后，通过配置主机代理或者操作系统自身的卷管理软件，将当前的系统盘，也就是CDP服务器上的虚拟影像，镜像回本地系统磁盘，镜像同步完成之后，重新启动主机。BIOS设为本地硬盘启动，启动系统之后，重新初始化CDP引擎，系统运行如初。这整个过程所耗费的时间很短，操作熟练的话，几分钟之内就能将一台崩溃的主机重新启动起来。

8) 二次改进模型

IO捕获模块运行在受保护的主机上，毕竟对主机资源有一定量的耗费，尤其是这个模块处于比较低层的位置，处理IO，延迟是首要考虑的

事情，此处产生瓶颈的话，就会对主机整体性能产生很大影响。应该考虑将其移出主机内部，放置到外部某处，实现对IO的捕获镜像而主机客户端只运行一个主代理模块即可，用于监控以及提供配置界面而且用于Snapshot管理等。主机IO路径的下一个关口就是FC交换机了，如果可以在交换机上实现一种类似端口镜像同时又比简单镜像更智能的写IO镜像的话，那么这个问题也就解决了。

幸好，两大FC交换机生产商Cisco和Brocade均已经开发了这个功能，分别称为SANTap Service和Brocade Storage Application Services。这两种功能在原理上是一样的，只不过分别由两家公司各自开发。其基本原理是：通过在交换机上运行一个智能的IO处理程序，其生成一个或者多个虚拟的Target LUN和虚拟的Initiator，受保护的主机所连接的LUN其实是这个虚拟的Target，这样主机针对源LUN的写IO便会被Virtual Target截获，然后通过Virtual Initiator将这份写IO同时发送给源LUN和镜像LUN，镜像LUN就是位于CDP服务器端的存储空间。CDP服务器需要通过FC链路向运行在交换机上的SANTap Service或者Brocade Storage Application Services注册所要保护的源LUN，注册成功之后，写IO才能持续的镜像到CDP LUN。

如图16-33所示为Brocade公司两款支持Brocade Storage Application Services的产品。

图16-33 支持BSAS的两款博科产品

如图16-34所示为两款Cisco公司生产的支持SANTap Service的产品

图16-34 支持SANTap Service的思科产品

二次改进之后的模型拓扑图如图16-35所示。可以看到块IO捕获虚拟设备已经不再捕获FC SAN上的磁盘的IO，而只负责捕获针对本地磁盘的写IO了。文件IO过滤层依然存在，因为需要满足文件级CDP的需求。

图16-35 二次改进模型

4. CDP所面临的重大技术难题：如何保证时序一致性

如果说Snapshot是普通照相机的话，NearCDP（准CDP）就好比是快速连拍照相机，CDP则是带移动侦测功能的摄像机。CDP之所以能够比Snapshot的回溯粒度更细，就是因为Snapshot是Copy on First Write，在Snapshot做成之后被覆盖过的Block，当再次被覆盖的时候，被覆盖的数据就不会再被Copy出来了，而CDP则依然会Copy，每多一个写操作就会多占用一份额外的IO仓库空间。再就是时间戳，因为用户决定CDP恢复的时候只会给出要恢复的时间点的数据，每个IO数据都需要记录时间戳，恢复的时候根据时间戳做统计。而Snapshot并不对每个IO都打入时间戳信息。再就是回溯之后的数据一致性问题，这个非常重要，下文讨论。

1) CDP和Snapshot数据一致性讨论

对这个话题进行论述之前，先介绍一下所谓“数据一致性”。举例说明什么是数据一致性：比如有某个程序正在运行，它的工作是计算 $1+1=2$ ，不停地计算，并将结果以“ $1+1=2$ ”的字串形式记录到一个文件中，并且假设这个程序在记录的时候是一个字节一个字节地写入的，比如先写入“1”，接着是“+”，接着“1”，再“=”，然后“2”。某时刻，程序运算这个加法完成了，立即开始向文件中写入记录，当刚写完“=”的时

候，存储系统触发了一个Snapshot，谁也没通知，而恰好在Snapshot之前一瞬间，文件系统也刚好将“=”这个数据成功写到了磁盘上。

在这份Snapshot中，对应这个程序生成的文件中的最后一条记录将会是这样的“1+1=”，这就是数据不一致的表现。不一致的数据会带来什么后果呢？某时刻，存储管理员将这份Snapshot影像做了Restore，重新挂载了该卷之后，这个应用程序启动，它需要将之前的数据结果读出并显示到显示器上，当它逐条扫描之后，直到最后一条它发现竟然只有等式的左边，结果没了。此时，根据程序的设计好坏会做出对应的反应，如果由于某些bug导致设计者根本没有考虑到这种只有左边没有右边的情况，很有可能造成程序崩溃。崩溃还算负责任的，如果遇到不负责任的程序员，不管三七二十一，凡是遇到这种情况，结果都是0，那就毁了，自己玩玩还行，拿到桌面上搞那可是要出大事情的。而设计良好的程序，就会充分考虑每一个细节，比如一旦发现这种不完整的算式，立即将其删掉，不再处理，并且报出具体错误。后者又称为一致性检查。数据库、文件系统均有这种机制。所以，直接在存储设备上随机生成的Snapshot，多数时候都是不一致的，Restore之后一般都要经过一层层的一致性检查之后才能最终被应用程序所使用。

还有一点很重要，即“数据一致性”和“数据丢失”的关系。数据一致，并不代表数据没有丢失，而数据没丢失并不代表数据就是一致的。如果理解起来困难，就看看上面那个例子，最后一条运算的结果很显然在那个Snapshot被Restore之后就丢失了，但是通过程序的一致性检查过程之后，不完整的记录被删掉，此时数据就变得一致了。所以说，数据一致并不代表数据没有丢失。那么数据没有丢失，是否就代表数据一定一致呢？非也，看看数据库系统就知道了，数据库对数据文件的每一笔Update都会记录到日志中，丢不掉，但是最终有意义的却是每笔

Transaction。每笔Transaction又由多笔Update形成，只有当最后一笔Update完成，并且应用程序向数据库提起Commit请求，数据库成功地将Commit点写入日志文件，此时这笔Transaction才算真正完成。如果在Commit点被写入日志文件之前，系统Down机，虽然这笔Transaction包含的每一笔Update都还在，数据都没有丢失，但是Commit点不存在，此时数据库在Replay的时候无法判断这些Update是否能形成一笔完整的Transaction，所以此时，数据就是不一致的，这些Update只能被Undo回退。数据没有丢失但是不一致的情况是比较理想的情况，通过Replay，总可以回溯到一致状态。

数据丢失

关于数据丢失，这里还需要理解一点。上文中说过，存储端随机生成的Snapshot无异于一份系统突然掉电时刻硬盘上的数据影像，那么有人问了，系统Down机的情况，随时随地都在发生着，而且根据上面的例子来判断，每次Down机都很有可能丢失些数据了？这个问题要从多方面来看，不仅涉及底层数据是否丢失，最重要的是系统后端与前端的沟通，以及终端操作员的判断和最终人为的介入。我们来演绎一个场景。

你去银行提现1万元，把银行卡交给操作员，操作员首先需要刷卡调出你的信息，然后会输入要提取的数值，然后会单击提取。此时，这个动作将会被传送至应用服务器端，应用程序将会在你当前的余额上减掉1万元，然后将更新的数据发送给数据库服务器，命令数据库服务器更新这条新记录，而且最重要的一步是需要对本次操作进行Commit，也就是提交操作。只有Commit之后，数据库服务器才会将这条记录写入日志永久保存于磁盘，而且是同步写入，不允许文件系统将数据缓存到内存而实际却不写入硬盘。当文件系统向数据库返回写入成功的消息

之后，数据库此时才会向正在等待Commit结果的应用程序返回Commit结束，此时应用程序便知道，记录百分之百已经写入了磁盘，掉电也不会丢失数据了。所以这时候，操作员终端机上才会提示存入成功，这时你才可以离去。如果在数据库成功向日志中写入了Commit点，正准备向应用程序通告Commit成功通知之前，数据库系统Down机，此时操作员终端机长时间没有收到回应之后将提示通信失败，未能提现。而此时，数据库确实已经将余额减少了1万。数据库服务器重启之后，系统恢复，操作员重新尝试提现1万元的操作，而此时操作员并没有关注账户中的余额其实已经相对刚才减少了1万，这次提现成功了。最终实际情况就是，你只拿到了1万，而系统却将余额减少了2万，你损失了1万元。如果是换成你存款1万元，按照刚才的场景演绎，操作员没有关注余额的变化，而进行了两次重复操作，给你多存了1万进去。

所以，数据一致性问题是所有层次都要关心的（VM，FS，APP），而数据丢失和误操作，不是数据存储层或者数据库层的问题，而是一种沟通问题和应用程序设计问题，同时，也是最重要的，就是人的问题。如果操作员仔细一点，就不会出现重复操作。然而话又说回来，计算机本来就是用来帮助人来实现人的需求的，如果需要让人来随时监控计算机所做的事情，如果每一笔都仔细核对，那没什么问题，但终究不是好办法。是不是存在人也无法探知的潜在数据丢失呢？很有可能，一笔记录丢了可以对一对，如果底层某个数据的丢失并没有通过某种渠道被业务层感知，或者丢失之后到被感知的时间过长以至于业务层也无法调查了，这是否就是死账形成的原因之一呢？答案是：人为因素+系统因素。

现在，我们对比一下Snapshot和CDP的数据一致性方面。先说说Snapshot的一致性。比如，摄影师给你拍照之前，你总得摆好姿势后才

能拍，也就是说你预先知道你要被拍了，已经摆出了最佳姿势了，此时应保持僵硬的笑容不许动、不许眨眼，摄影师按下快门。这就好比 Application Aware Snapshot和FS Aware Snapshot，即能够感知应用程序和文件系统对数据的操作此时已经处于一个完整一致的状态，缓存中没有留存的只向磁盘中保存了一半结果数据的运算，此时立即对底层卷触发一个Snapshot，此时的卷影像上的数据就是一致的。即，用这个Snapshot进行回溯/Restor之后，应用程序将对应的数据读出之后，不会产生异常结果。

这种方式需要在主机端安装一个Agent来负责将主机的状态告知存储系统从而在正确的时机生成Snapshot，或者由人为干预，在生成Snapshot之前将应用程序退出或者静默。如果你不想受到约束，要求摄影师随机拍照，拍完之后再一张一张挑选姿势恰好的，那么也没有问题，这就对应了在存储端随机生成Snapshot，乱放枪，打着赚了，打不着也不浪费什么，但是要查看哪个Snapshot是一致的，哪个是不一致的，那可就难了。Snapshot毕竟不是可以拿在手中端详的照片，除非另外部署一套应用系统，一个一个地挂载Snapshot卷，一次一次地启动应用程序判断，这也属于无事生非。至于抓拍偷拍，目前的存储系统可没有这个能力来直接感知应用层什么时候处于一致状态，再说也用不着这样，做个Snapshot还得藏着掖着不让人知道。

再来说说CDP的一致性。看照片是看瞬间的姿势，看电影则是看情节，电影的情节是一段一段的，漏下一段没看的话，就感觉很不爽了。此时你需要将播放器的滑动条往回拖，拖一拖，看一看，再拖一拖，再看一看，直到你有记忆的地方为止，接着往下看。这个场景恰好对应了糊涂兄让小管恢复数据的那个场景，小管用二分法尝试了N次才找到那份文档的最后时间点的影像。但即便是这样，我们仍然无法确定哪个时

刻的影像是对于应用层一致的，因为底层的处理根本就无法感知上层的逻辑，除非一次一次的尝试，但是这样做却是费时费力而且也失去了CDP的意义。所以说，CDP虽然在回溯时间粒度上比Snapshot强大，但是在保证数据一致性方面，却不如Snapshot。在后面的产品介绍章节中我们就会看到，存储产品厂家是深刻意识到这一点的，所以它们结合了CDP和Snapshot的优点，将二者整合起来使用。

利用CDP随机回溯到任何一个时间点之后所得到的数据影像，很大几率上都是不一致的，此时，影像被挂载到主机端之后，一定要经过一层层的一致性检查。一致性检查会在下面几个层次上依次进行：VM卷管理层，文件系统层，最后是应用层。首先是VM层，VM是一个程序，它自身需要向硬盘上写入MetaData元数据用以记录虚拟换和物理卷地址的映射关系，如果这些数据变得不一致或者丢失，轻则无法挂载（这也是为何Down机经常导致操作系统无法启动或者不认卷、无法挂载卷的原因），重则卷边界错乱以假乱真。所以VM在挂载某个卷之前一定要对其进行数据一致性检查。VM层之上是文件系统层，与VM同样的道理，文件系统MetaData同样要经过一致性检查的步骤，检查完成之后方可被挂载。最后就是应用层了，应用层逻辑上的不一致只能够靠应用程序自身解决。在一层层一致性检查完成之后，系统方可恢复工作。

2) 一致性组技术

设想这样一个应用程序，它需要将数据存放在多个不同的卷中，而且这多个卷中的数据有相互依存关系，哪个卷中的数据发生不一致或者丢失，其他卷中的数据也就不再可用。要对这样的环境做Snapshot或者CDP数据保护，需要考虑额外的步骤，即必须保持这多个卷在相同的时刻同时被触发APP&FS Aware Snapshot，保证APP和FS层面的一致性。

那么对于CDP来说，由于CDP的先天优势，即IO时间戳，既然每个卷的IO都被打入时间戳，那么只要给出一个时间点，CDP所遍历出的卷影像，其上的IO写入时序一定是可以保证严格按照时间的行进排列的。比如，T1时刻应用向卷1写入了数据，然后T2时刻又向卷2写入了数据，当用CDP恢复卷1到T1时刻之后，卷2则必须也恢复到T1时刻。虽然CDP不能保证任意时间点的上层应用一致性，但是它起码可以保证底层的一致性，然后靠应用的Replay过程来完成最终一致性恢复。

一致性组技术对于Snapshot和CDP来讲其实并不需要多么复杂的额外设计，只需要将相互关联的多个卷组成逻辑组，做Snapshot的时候保证同时触发即可，CDP则更是无须做什么本质改变，只需要在程序界面中告诉操作员哪些卷在一个组中即可。

而真正需要大动干戈的是远程异步数据复制容灾系统，多个关联卷的数据一致性问题的解决需要引入更复杂的一致性组技术。在DR系统中，“一致性组”将有两个含义，下文以及本书后面章节会详细介绍一致性组技术的细节原理。

3) 关于CDP的回溯粒度问题的论述

不少人认为CDP既然连续捕获了每个写IO并且打入时间戳，那么CDP的回溯粒度最小可以达到一个IO的级别。这种看法是错误的，持有这种观点的人忽略了一个问题，即时间的最小粒度和上层的IOPS数。本书第3章中的3.1.3小节中曾经得到结论，即时间是没有最小单位的，在目前的认知领域中可以认为其是无限连续的。

设想这样一种情景，某CDP设备所接收到的写IOPS为每秒10000次，也就是说平均每毫秒会有10次IO。如果这个CDP设备的恢复粒度可

以达到一个IO的级别，即分辨出这1毫秒之内发生的所有IO的先后顺序，那么它的额定时间戳粒度就必须至少设置为0.1毫秒级别。如果收到的写IOPS每秒100万次（当然目前现有存储系统可以达到这种级别，CDP产品就更别提了，这里只是理论推导），则要达到回溯每个IO的粒度，时间戳设定粒度就要设置为微秒级别了，以此类推。时间戳粒度越细，维护的开销就要越大，耗费资源越大，效率也就越低。但是如果非要实现单个IO级别的区分，也不是没有办法，如果对每个IO标记一个自然序号而不是时间戳，那么就可以完全分开每个IO了（这种方法同样可以用于容灾技术中的一致性组技术中，参考本书后面章节）。此外，IO序号的功能不仅仅是区分每个IO先后，而且它的另一个重要作用则是对乱序接受的IO进行重排和丢包监测（见容灾一章中一致性组一节）。由于数据包在网络上传输的时候有可能是乱序到来的，如果将乱序的IO按照顺序追加到CDP日志链末尾，那么所保存的IO也是乱序的，这样不管怎么搞都不可能做到时序一致。

话又说回来，既然CDP随机回溯后的数据一致性本来就几乎不可能保证，没有人能够判断在哪两个IO之间切开时数据是一致的，所以以每个IO为回溯粒度，不仅没有必要而且也没有意义。通常时间戳粒度设置为毫秒级别已经足够。

注意： 只用时间戳而不使用自然序号来标记IO的情况下，不能够保证单个IO的时序一致性（见本书后面章节对一致性组技术的讨论），所以生成的LUN影像也将会是时序不一致的，而这种不一致是最严重的一种，可以导致数据库类应用程序甚至操作系统都无法启动。所以，一个CDP系统设计是否考虑了数据一致性，是非常重要的考量指标。所以，同时使用序号和时间戳才是解决数据时序一致性错乱的有效方法，序号用于区

分单个IO的时序，时间戳则只用来定位用户给出的大致的影像时间点。

然而，即便是同时使用时间戳和序号也只能够保证全局日志连模式（见下文）的CDP。对于每个块均对应一个日志链的CDP系统，由于块与块之间的日志链之间互相独立，当用户要求某时间点的卷影像的时候：CDP系统就需要在每个块的日志链中找到用户所要求的这个时间点所对应的块，此时问题就出现了：每个块日志链的序号如果都各自计数的话，那么就无法分辨块与块之间的先后顺序了（不像全局日志链）。此时的解决办法就是全局计数，不管所接收到的IO目标是哪个块，每接收到一个IO计数器就加1并且将序号保留在对应块的日志链中。

同时使用IO序号和时间戳必定会耗费不少资源，为了提高效率，有些厂商定时在每个块的日志链中做标记。

理想：有若干条豆腐传送带，每条传送带上接收到的豆腐速率不同，形状也不同，间隔也不同，如果要区分所有传送带上的所有豆腐的到达先后，是很复杂的。但是如果降低粒度，比如每秒记录一次，保证秒与秒之间肯定是先后排列的，但是一秒之内所接收的豆腐无法分辨其先后顺序，这也不失为一种变通方法。所以，在这些传送带上方放一把闸刀，每秒切下一次，这样豆腐不管是什么形状，以什么速率到达，豆腐块之间间隔多少，只要闸刀一落下，所有传送带上的豆腐此时都被定型，每秒定一次型。当数据回溯的时候，只能恢复到每次定型的时候。这个思想就是“一致性组”的思想。（一致性组的具体内容可以在容灾一章中阅读。）

CDP即便是从理论上讲，也不可能做到每个应用程序的写IO都不

遗漏。因为有缓存机制的存储系统层都有一种所谓Delay Write的机制，比如文件系统层，某时刻应用程序要求更改文件名A为B，这个要求被文件系统收到，文件系统会读出这个文件的元数据，然后将对应的部分更改。但是更改之后的dirty数据块先不急于写入磁盘，会缓存一段时间，一旦在这段时间内，应用程序再次发起请求，要求将B再改为C，文件系统接收到请求之后会发现这个文件的元数据所对应的最新数据块已经在缓存中的某个dirty块中，则文件系统直接再次更改这个dirty块使其二次dirty，直到文件系统将dirty块写盘之前，任何再次的更改请求都会直接更改。这种情景为“写命中”的一种情况（其他情况见本书后面章节）。这样，应用程序的多次写IO请求，在底层却只表现为文件系统的一次写IO请求，如果CDP是作用在文件系统及之下的层次，那么这样的CDP系统就不能截获每个应用层的IO了。

能做到真正意义的应用层一致性保障的以及最细粒度的CDP其实就是基于应用层的基于Transaction日志链记录的CDP系统，每一笔改变都不会丢掉，而且占用存储空间少，只不过针对应用层的CDP需要针对每个应用来开发，不像底层CDP一样具备通用性。在此是否可以大胆想象一下：今后可能会出现类似VSS公共快照服务一样的公共CDP日志中间件。

通过记录日志的方法很容易做到一致性保证，但是对于使用分块日志链的卷级的CDP，由于为了保证任意历史版本可以瞬间挂载读写，不可能采取全局日志链（见下文）的方式，必须保证每个块的历史版本都可迅速定位寻址。这样的话，如何在这些众多的块日志链中迅速地抓出一份具有时序一致性的块链出来，成了一大技术难题。下面我们来看一下CDP是如何攻破这个技术难题的。

4）自演绎的一招——疾风镂月斩

从上文中的“切豆腐”思想，作自己推演出了一个保证CDP回溯时顺序一致性的模型。说明：此模型完全为个人演绎，如与某专利或者厂商实现雷同，纯属巧合，请勿对号入座。

在如图16-36所示的二维象限中，横轴是时间轴，纵轴上的每个点代表某个LUN或者卷中的一个Block/Sector/Cluster，具体粒度是块/簇还是扇区可以人为选择，一般使用Block作为最小单位。这里只示意了8个块，并不表示实际情况。

图16-36 疾风缕月斩关系模型

在没有引入CDP之前，这个LUN是没有时间轴的，此时整个LUN就是一个线性的由所有Block所串起来的一维直线，不可回溯。当引入CDP后，时间轴被拉开，将一维直线拉成了二维的平面，这个平面就是图16-36所示的二维象限。这个象限中的每个点 $B(T_n, B_n)$ 表示卷中对应的Block在某时刻的内容，这个象限的本质其实就是一个CDP分块日志链。每个块每次的写IO内容都被记录了下来，每个块在每个时间点都有它对应的内容。 T_0 时刻表示CDP引擎启动之后对应的卷影像时刻，即尚未有任何变化的时刻， T_∞ 时刻表示当前最新时刻。采用CoW模式的CDP系统中，当前时刻的源卷影像就是最新的卷影像。 T_n 表示任何一个介于 T_0 和当前时刻之间的时刻，即任一历史时刻。

F这个函数关系没有规律可循，因为IO顺序、间隔都是无规律可循的。正因如此，可以看到所有块日志链中的历史块的存在形式是凌乱而没有规律的，除了 T_0 时刻的块纵向链之外。因为 T_0 时刻是一个确定的时刻，CDP系统做好准备接收卷IO的那一刻，就是 T_0 时刻。但是随后的时间中，IO可以在任意时间点到达任意块，所以从 T_0 之后就再也没有规律可循。只要对应的块有写IO，那么第一个被CoW出来的块就是 T_0 时

刻的历史块，所以可以看到这个时刻的纵向块链是整齐排列的。另外，由于B3这个块从未被修改过，所以它的日志链为空，当前的B3块的内容等于T0时刻的B3内容。

上文中多次说过，面对高IOPS数的情况下，系统可能对一个块日志链中先后来到的多个甚至上千个被CoW出来的历史块打入同一个时间戳（或者使用时间戳+全局序号法来避免这个问题，但是消耗资源太大），那么此时整个系统就不可能分辨出纵向的块与块之间的时序先后性了（横向依然可以分辨，但是没有意义）。上文也提到过，解决这个问题一个变通方法就是定时地在所有块日志链中做标记以划清时间线。

那么大家来看一下，何时、何地、怎么来做标记呢？如果这个做标记的动作发生在后台，即前台源源不断地将历史块存入日志链之后再做标记的话，一定行不通，因为此时你已经无法分清到底该在哪两个块之间切开了，具有相同时间戳的块随处可见，此时再做标记已经晚了。那就只能在前台做标记，也就是说在被CoW出来的历史块尚未写到日志链之前，用一把闸刀纵向地在所有块日志链上挥斩一下，被斩到的块便被做上对应时刻的标记，之后这些已经标记的块便被写入对应的日志链。以不同的频率挥刀抡斩，就可以做到不同粒度的CDP回溯。

我们假设CoW出来的块会按照图示的顺序和间隔到达，如图16-37所示。Ta时刻系统做了一次斩断（对应到专业说法是“采集”或者“采样”），其中B0、B2、B7这三个日志链都斩中了块，而剩余的日志链斩空。对于斩中的块，系统对其做时间线标记然后追加到日志链末尾。对于斩空的日志链，可以认为当前时间点对应的块内容与本日志链下一个即将到来的块相同，此时系统可以沿着日志链的上游方向监测，如果在线程监测的时间段内恰好有刚被CoW出来的块准备存入该日志链，则线

程将其标记纳入本次斩断的时间线；如果在监测时间段内没有新块被迫追加到该日志链，那么线程需要从源卷将对应的块内容读出（注意读取对应的块时需要考虑线程竞争互斥，必须锁定该块防止新数据将其覆盖），然后做标记纳入本次时间线，然后追加到日志链末尾存放。

图16-37 Ta时刻挥刀抡斩

然而，对于那些更新频率很小的源卷块来讲，每次斩断均需要读取源卷，每次读出的内容都相同，除非对应块被更改，这样的话日志链中将会有很多冗余数据并且耗费额外的读操作。对于这个问题，可以使用另外的方法来解决，比如对每个由于斩空而从源卷读出并追加到日志链的块，除了做时间线标记之外，再做一个特殊标记来表明它的特殊性，这样，当下一次斩断时刻到来的时候，系统检查日志链末尾，如果末尾的块被标记为这种特殊块，那么系统只需要完成日志链上游巡检（其实就是向CoW线程进行查询）阶段即可。如果巡检完成之后未发现已被CoW出的块，那么本次斩断时间线可以直接指向日志链末尾的那个块而无须再次复制，节约了资源。

在图16-37中，标记为C的块意味着此块是从当前的源卷中复制出来的，可以看到由于B3、B4、B5这三条日志链在挥刀抡斩的时刻并没有CoW出的块到来，所以Ta时刻对应的块的内容就是源卷对应块的内容，因此系统从源卷将这三个块读了出来追加到日志链末尾存放并做时间线标记和特殊标记。被标记为W的块表示相关线程沿日志链上游方向巡查时一定会遇到该块，所以系统会等待（W）该块的到来，届时会将其作时间线标记（不做特殊标记）。这里的所谓“等待”其实并不是真正的去等待，而是采样线程主动地查询CoW线程去探测是否已经有CoW出的块正等待写入日志链，如果有则去获取，没有则按照上文所述的逻辑继续处理。

依此类推，在 $Ta+b$ 和 $Ta+c$ 时刻，系统又做了两次斩断。可以看到，B3、B4、B5三个块被公用了，节约了IO资源，如图16-38所示。

图16-38 定时挥刀抡斩

我们对图16-38做一些抽象工作。对于每个时间线，沿着时间线所跨越的块的右边界划线，如图16-39所示。

图16-39 划定时间线

然后，将凡是右边界不贴线的所有块进行镂空，剔除它们，只留下被时间线跨越的块，如图16-40所示。

图16-40 镂空不可回溯的块

然后我们再将被时间线所跨越的块也全部删掉，如图16-41所示。这样，这个象限中只剩下了我们所画的线条。可以将这个线条称做“回溯线”。只有右侧边紧邻回溯线的块才可以被回溯，也就是图16-40中所包含的那些块。

图16-41 回溯线

回溯线是纵向延伸、横向上有所浮动的线，理想情况下，每次斩断所生成的回溯线不交叉不重合，便于分辨，但是一旦遇到共享块，那么多条回溯线便会有所重合和交叉。回溯线是在纵向方向上将这一时间点的所有块的内容连接成为一个完整的卷影像的脉络线。如果利用快刀斩乱麻的方式来鲁莽地将日志链切分，那么得到的虽然是个光滑表面，但是这个光滑的表面是毫无用处的，不能保证时序一致性；而利用疾风镂月斩所生成的虽然是一条曲遛拐弯的坑洼表面，但是依附在这个表面上的数据块都是带有时序一致性的。

每次进行回溯的时候，系统会根据用户所指定的时间点来寻找系统中离用户给出的时间点最为接近的斩断点，然后将数据回溯到此处，这样即可保证整个卷所有块之间的时序一致性了。

另外，本招之所以被称为“疾风镂月斩”，不但是因为其像疾风一样迅速，而且还因为它可以将日志链中不可回溯的无用块剔除掉以节省空间。

5) 改进之后的招式——迅雷幻影手

疾风镂月斩在斩断点之间保存了大量的无用历史块，写入这些块需要耗费额外的IO资源。与其先写入再镂空，为何不干脆就不要将这些明知后来将要被镂空掉的块写到日志链中呢？

思考： 想象一下，有成千上万串的IO在不断下落，速度非常快，此时有一只手，它能够以迅雷不及掩耳的速度将某个时间点的某个固定横向方向上的正在下落的IO抓取出来，没被抓取的IO块随后就被湮灭了，只保留那些被抓取出来的块。

是的，在疾风镂月斩一招中，负责斩断的线程和负责写日志的线程之间沟通的不够，导致效率低下。如果让斩断线程作用于写日志线程之前，日志写线程只写入那些被保留的数据块并且做相应标记，这样就可以大大提高效率。

且慢！仔细考虑考虑。如果这样做的话，那么在两次采样的间隙中所发生的更改块就不会被保留，可能在某个采样间隙之中，某个块已经有了多次更改，而都没有被记录下来。这样，当下一次采样点时，如果恰巧没有采到针对这个块的更新块，而上一次采样时的块是从源卷复制

而来的，那么本次采样依然会将回溯线贴附到日志链结尾的上一次采样点对应的块中。而这样显然是错误的，因为这个块在之前的间隙中已经有了多次更新，而晚于这些更新时间点的采样却错误地认为采样点对应的块与上次采样相同。

所以，迅雷幻影手这一招，用不得！只能先留底，所有IO一个都不能少，后删除，后台处理，先斩后奏！

6) 一次修炼之后的新套路

之前的招法都是基于CoW模式来设计的，CDP服务端不停的CoW，这样的话如何保证性能？由于产生额外的2倍写惩罚，相当于IOPS和带宽耗费变为不启动CDP时的3倍！这怎么可以让人接受呢？必须无条件转向RoW模式，有再大的技术难题，也要攻克！我们之前曾经讨论过CoW与RoW方式的对比，RoW耗费IO资源，但是后续的所有访问会一直持续地计算资源，但是这个结论对于CDP环境就不成立了，主机发起的读操作不会落到CDP镜像卷上，而是落在了源卷，这样根本不需要CDP来处理什么。既然这样，如果在CDP环境下使用RoW模式，那么将不会影响到主机IO。RoW模式虽然不适合大部分快照设计，但却非常适合CDP设计！

此外，可以看到在疾风缕月斩招法中，首次采样时如果遇到某个块从来没被CoW过，那么需要从镜像卷中将其复制出来放到日志链中，而第一次采样时基本上大部分的块都是尚未被覆盖过的，那岂不是等于需要复制出镜像卷中大部分的数据块到日志链中？几乎等于再将镜像卷又镜像了一份！在首次采样的短短时间之内需要做这么大量的事情，从空间和性能上都不能接受，是否可以只用指针来指向镜像卷对应的块地址呢？其实，首次复制这个动作是有苦衷的。

试想一下，一旦CDP引擎（CoW或者Row引擎）发生故障，中断工作，而源卷可不管你CDP是否故障，它还是在不断地被写入。

那么此时有两种处理办法。第一种处理方式是主机端的IO镜像器停止向CDP服务端发送IO数据，等CDP恢复之后再次发送，如果这样的话，从故障到恢复这期间的数据变化就无法被同步到CDP上的镜像卷了。就算CDP恢复之后，再次采样，那么所生成的历史时刻影像就是不一致的，因为缺失了一段历史，在采集之前必须补回来才可以。解决这个问题的办法是在主机端IO镜像器设计一个位图用于追踪断开与恢复之间的数据变化，当恢复之后，首先利用这个追踪位图将这些变化的块不断地传送到CDP端覆盖到镜像卷，同时不断地记录正在发生的源卷改变，之后不断地追赶，最终达到收殓状态，双方严格同步，然后再启动采样进程开始采样。但是恢复之后的采样进程在首次采样时也依然需要将镜像卷中对应的块复制出来放到日志链中，这又会占用大量的存储空间。

提示：这样做的好处是：CDP故障之前的历史版本都还可使用，因为当时首次采样时做了复制留底而不是用指针指向镜像卷，所以，CDP故障期间所丢失的历史时刻数据仅仅是故障到恢复期间这段时间的历史。所以，如果首次采样时只用指针指向镜像卷对应的块，那么当镜像卷上的某些块发生覆盖的时候，指针就无效了，之前的历史版本数据也就都随之湮灭了。

第二种处理方式是：当CDP引擎发生故障时，主机端IO镜像器依然源源不断地将写IO发送过来，此时这些IO会直接覆盖镜像卷上对应的块，因为此时由于CDP故障而做不了拷出或者重定向动作了，如果当初没有首次采样复制而只是用了指针，那么之前的指针与现在的实际数据的时序一致性将无法得到保证了，因为镜像卷已经受到了“污染”。所

以，之前所有的历史版本数据此时也同样都将不再可用。

如果使用指针，那么故障之后当CDP引擎恢复时，就必须从头开始，之前所有保存的历史数据版本都不可用。而如果首次采样时复制一份基线版本出来，那么就算CDP引擎暂时停止工作，也只是导致从停止那一刻到恢复那一刻起这之间的历史时刻不可回溯，再往前或者再往后的版本都可以回溯。另外，不管用指针还是首次复制方式，当CDP引擎恢复之后，需要以当前时刻的镜像卷为基线版本重新开始采样，这也就意味着不能与之前首次复制的块公用了。这样做显然是非常浪费空间的，可以估算日志卷至少是源卷的一倍大小，这是至少。要真正跑起来，恐怕得1.5倍而且还得期望CDP引擎不要中断，中断了再恢复如果还想保留以前的历史版本的话那就得2倍起了。

鉴于首次复制方式太过浪费空间，也无法让人承受，只好退而求其次，一旦CDP故障或者中断，那么在CDP恢复之前，用户可以选择将几个关键时间点的数据影像备份下来，然后再恢复CDP运行。恢复之后，所有历史版本数据将被删除重来，但是CDP服务端的镜像卷可以保留，主机端的IO镜像器可以在CDP中断工作期间记录变更位图，当CDP再次启动之后，主机端可以只将变更的块再次同步过来即可。

经过上面两层的修炼之后，我们有了结论：必须使用RoW方式，必须抛弃首次复制的做法，改用指针。我们下面看一下RoW+首次指针模式下CDP引擎的工作流程。

如图16-42所示为RoW+无首次复制模式下采样示意图。

图16-42 RoW+无首次复制模式下采样示意图

T0时刻的块链（卷），也就是图中所示的CDP引擎开始采集的那一

瞬间的基线卷版本。当采集尚未开始时，所有对这个卷的更新会直接覆盖于其上，不产生RoW动作。一旦采集线程开始工作时，那一瞬间的卷版本即被冻住，随后针对这个卷任何块的更改内容均被RoW追加到每个块的日志链中。每隔一段时间，采样线程触发采样。

T1时刻触发一个采样点，采样线程首先向日志链写线程发起查询以确认是否当前时刻存在即将被RoW入日志链的数据块，也就日志写线程已经对外面返回ACK写成功信号但尚未写到日志中的数据块。如果得到日志写线程的回馈说没有，那么采样线程直接对日志链尾部的块做标记，将其标记为本次采样的命中块。

本例中，对于B0、B1、B2、B5、B6这5个块的日志链，采样线程并没有查询到有即将被写入的RoW块，并且在日志链尾部也并没有之前的RoW的块（本例中T1时刻这几个块的日志链为空），所以采样线程直接指向T0时刻卷对应的块上（在日志链末尾生成一个指针记录）；对于B3、B7的日志链，采样点时恰好采到了日志链末尾的一个RoW过来的块，标记之；对于B4的日志链，采样点时刻并未发现日志链末尾有RoW的块，但是通过查询日志链写线程，得到回馈，正有一个RoW的块要被写入日志链，所以采样线程等待日志写线程成功将这个块写入日志链之后，对其进行标记。至此，T1时刻采样完成。

T2时刻再次触发采样。同样，采样线程首先查询日志写线程得到有哪些块的日志链将要有RoW的块被写入，对于那些有RoW块即将被写入的日志链，采样线程就等待其写入；对于那些没有RoW即将被写入的日志链，采样线程直接对日志链末尾的块进行标记，标记为本次采样的命中块。如果遇到尚为空的日志链，则在日志链尾部生成指针指向T0时刻源卷对应的块。

之后的所有采样点都依此类推，执行同样的过程。同时，系统可以启动一个后台进程，专门对日志链中没有被采样点采中的块进行镂空操作，将其删掉以节约空间。删掉之后就变成了如图16-43所示的样子，每条采样线粘住的块链就是这个时刻这个卷的历史影像了。经过再次抽象，将粘住的块也腐蚀掉之后，只剩下了抽象的回溯线，系统只能回溯到这些线所代表的时刻以及块链，也就意味着只有右侧紧贴回溯线的块才可以被回溯。

图16-43 先斩后镂及镂空后显现出来的回溯线

CDP引擎必须作用于整个CDP服务端的总IO Queue之下，缓存之前。即在Queue这个水渠处做掐断，这样才可以分辨出每个IO的时序先后然后严格按照时序下发到日志链中，从而让采样线程中相关时序逻辑的作用结果达到预期。如果等到所有IO都进入了缓存，然后再将各个块内容下发到日志链，那么此时就像大海捞针一样，无法分辨出时序，所下发的IO都是乱序的。日志写线程为了保证每个IO被按照顺序写入日志链，也要耗费一定资源，比如在写入数据之前，需要把所有待写入的IO数据块做成一条按时间先后排列的链，然后依次下发，这样是非常低效的，必须进行流程优化。

进一步优化流程：在总Queue的尾部处执行Suspend暂挂，当然暂挂期间，主机的IO依然可以下发，Queue不断充满。当掐住点之下的IO全部写到日志链中之后，采集线程立刻做采集，采集完毕，回溯线生成之后，总Queue解除暂挂。这样，数据按照批次来写入日志链，这一个批次之内的IO不需要分辨时序，但是批次与批次之间是严格按照时序先后发生的，每一批次的IO被写入日志链之后，再进行采集。这样做，不但可以简化日志写线程的设计以及流程，同时也简化了采集线程的设计与

流程，因为采集线程此时不需要沿着日志链向上游巡查了，也就是说采集线程在采集时不需要向日志写线程进行查询操作了。

此外，除了在CDP服务端存储阵列的总IO Queue处实现暂挂外，还可以定期的在主机端相应的层次，比如文件系统处、各种应用程序处实现一致性刷盘暂挂，待CDP采集线程采集标记完后解除暂挂，这样就可以实现完全应用层的一致性了，这些采样点便是应用层一致的。不过这种采样不可多做，因为会对应用产生影响，定期做即可。这样，在日志链中既可以保证若干份应用层一致的影像，又可以保证一致性层次较低的大量其他历史时刻的影像。

批量IO以保证时序一致性的做法是个通用做法，已经用于数据远程复制技术中。这种技术又被称为“一致性组”技术，关于这一技术更详细的阐述请见本书后面的章节。

经过这样的流程优化之后，系统就可以利用有限的资源生成更高频率的采样，降低CDP的回溯粒度到秒级甚至更低。

7) 二次修炼之后的最终招式——雕心镂月幻影斩

之前所列出的招式，无一例外都需要在块日志链中做动作，标记对应的块。但是仔细思考一下之前的模型，每个块都有一条物理上的日志链，那么一开始要为每个块的日志链分配多少空间呢？如果某个块频繁被更新导致其日志链迅速充满，此时如何解决？另外一个问题是，每次采样时需要对所有的块做标记，需要做大量的写入动作，而且这些写入都属于随机IO类型，那么此时如何保证IO性能？针对以上两个问题，虽然可以有各种技术手段解决，但是其效率将会是非常低的，尤其是IO效率上，将会使整个系统运行缓慢。

如图16-44所示为分块日志链与全局日志链的物理结构比较，很显然，对分块日志链的写入动作将会非常耗费资源，但是写入全局日志链就不用考虑那么多了，所有分块的RoW的IO只要追加到日志链末尾即可。但是这样做又不利于快速寻址，也就不可能实现瞬间挂载历史时刻的影像。那么有没有什么办法将这两种模式的优点相结合呢？

图16-44 分块日志链与全局日志链

答案似乎已经浮出水面了。如果把分块日志链的这个框架全部搬到内存中，做成一张分块日志链表，而底层实际数据的存储使用全局日志链的模式来存储，在分块日志链表中不保留实际数据块而只保留数据块在全局日志空间的地址指针，那么就可以完美地解决效率问题了。

如图16-45所示为全局日志链存储结构与分块日志元数据链结合之后的新模型。通俗一点说，每个块的RoW的IO到来之后全部顺序追加到日志空间末尾，属于一种完全连续IO，同时在分块元数据日志链中将每个RoW块被写入到日志空间内的地址指针记录下来并且追加到每个块日志元数据链的末尾，由于这个动作是在内存中完成，不需要对磁盘进行额外IO操作，所以可以保证性能。

图16-45 全局日志链存储+分块元数据链表

在每次采样点时，系统也只会分块元数据日志链中做标记，而不需要去日志存储空间内相应的块上做标记了，也避免了无谓的磁盘IO消耗。采样点之间的无用块可以在后台将其镂空掉以降低日志空间的耗费，同时分块元数据链中的对应这些无用块的指针也可以被删掉以降低元数据链的容量和复杂度，加速寻址速度。RoW线程维护一个对日志空间的Space Bitmap来充分利用被删除的无用块后所腾出来的空余空间。

分块元数据日志链相当于给无法直接迅速寻址的、顺序保存的RoW数据块空间创建了一份类似于文件系统思想的、可迅速寻址的元数据链。全局日志链底层的每个RoW数据块都是完全随机排布的，没有按照每个块来划分，但是这不重要。正像文件系统中的文件在底层也可以是完全随机分布一样，只要有一条清清楚楚的元数据链图纸，按图索骥就可以将对应的块串成一条按照历史时间先后顺序排列的链了。

相对于之前的“疾风镂月斩”所用的带内元数据记录方式（直接将日志链按照分块来排布而不是全局排布，并且直接在分块日志链的数据块上增加标记），“雕心镂月幻影斩”则属于一种带外元数据记录方式，其好处是对实际数据的排布方式没有限制，可以按照最优的模式来排布。

分块元数据链表虽然存放的全部都是地址指针以及相应的标记，容量远小于实际数据日志，但是其容量也是不可忽视的，需要为其提供一个专门的存储空间。为了保证性能，元数据链全部载入内存，所有的更改在内存中进行，刷盘时间可以适当提高而不必频繁刷盘。频繁刷盘意义不大，因为如果CDP服务器发生故障、宕机之后，需要做重新同步，同步之后，之前所有的历史时刻影像均会被抹除。

挂载某历史采样点的影像之后，针对影像中某个块的IO操作，均需要查询分块元数据链来寻找对应采样点回溯线上对应的块的地址指针，然后通过这个指针到全局日志链中将对应地址的块内容读出并且返回给主机端。如果遇到针对这份虚拟影像的写操作，那么就像之前一样，再次对这些二次写操作进行RoW，形成新的日志链，也就是日志链上某个历史时刻的分支日志链，也就相当于历史回退之后，从回退点再次向前发展，将所有的发展用相同的方法记录下来，从而使得对虚拟影像的变更也可以回退。或者选择将当前的虚拟影像同步到一个新存储空间，比如主机直接从这份虚拟影像启动，然后将这份影像同步到本地启动盘，

同步之后再切换到本地盘启动，这就是一个典型的本地操作系统或者物理故障之后的迅速恢复过程。

从上面的模型不难看出，对CDP虚拟影像卷的读IO操作为完全随机的IO类型，因为历史时刻的块是完全随机分布在全局块日志链中各个位置的，这也会导致性能很低，请意识到这一点。但是对其的写入操作由于会被不断地追加到日志链末尾，属于一种连续IO类型，所以上层的写IO不管原生状态有多么离散随机，到了地层全部变为连续IO，从这种角度来看，CDP服务端至少不会被随机写IO所拖慢，反而还可能比主存储的响应时间更快。不仅对虚拟影像的写入操作时如此，所有CDP所接收的写IO在底层均被连续化处理，充分屏蔽了写IO的原生随机性，这样就可以大大降低随机IO对系统性能的影响，随机度完全被体现在分块元数据链中。

8) CDP和快照的生态关系

我们回头看一看16.2.3节中对快照底层的演绎，发现CDP和快照其实本是同根生，快照是用一个全局IO仓库卷和多份映射表来维持块映射关系，而CDP则是用多份日志链和多个回溯线来维持块映射关系。CDP中的回溯线相当于快照系统中的块映射表，每次生成一份快照都需要新建一份新映射表，同样，每次生成一个可回溯点，也需要生成一条回溯线。快照是个照相机，而CDP则是个摄像机，摄像机与照相机的区别就是摄像机每秒可以过24张胶片，生成每秒24帧的可以骗过观众眼睛的影片。电影尚且有每秒24帧的粒度，那么CDP是不是也可以使用某种粒度来满足用户需求呢？一个道理，没有必要记录到自然界的每一个微小的位移，就像没有必要记录每一个IO一样。

CDP的主要价值还是体现在本地和远程容灾，CDP自身这个技术只

是一个幌子。两分钟做一个快照，保留30份或60份，也就是可以以两分钟的粒度回溯最近一个或者两个小时内的数据，或者比如半分钟一次快照，保留一个小时的，再早的通过磁带等其他备份介质来恢复，这样算不算CDP？也差不多。

思考：以特定频率生成回溯线的CDP系统，就是所谓Near CDP。本质上其实就相当于高频度的快照。所以说，我们经过这场游历之后，最终发现：CDP是快照的进化形态，粒度的大小是评价它是快照还是Near CDP或者是Pure CDP的唯一标准，粒度并没有一个界限，正因如此，CDP和快照之间是一个随着粒度变化而连续变化的形态。CDP相对于快照并没有本质上的进化。所以，目前市场上的CDP产品本质上都是Near CDP。如果有人宣称他可以回溯每个IO做到真正的Pure CDP，基本上是在忽悠你，因为就算底层实现了回溯每个IO，用户也不可能一个IO一个IO的来回溯，没有实现的必要。

9) CDP产品对应用层一致性的解决办法

上文中给出了解决IO时序一致性的方法。然而，仅仅保证时序一致性已经无法满足用户需求了。保证了时序一致性，与突然宕机没有本质区别，突然宕机之后硬盘上的数据也是具有时序一致性的，但是对于数据库等应用来讲，宕机10次，大概会有2次导致数据库无法启动，需要从备份中恢复。所以，CDP系统最好可以提供应用层一致性解决办法。

要解决应用层一致性问题，必须与应用层来交互，也就是说必须在主机端安装一个代理程序。有两种办法来解决：第一种是代理程序监控文件系统的Flush点，在每次Flush完成之后，立即通知CDP系统，CDP引擎便在当前日志链中对当前时间点进行标记；第二种办法则是直接与

应用程序进行交互，比如代理程序向对应的应用程序发送Clear-Up/Flush请求，完成之后便通知CDP引擎在日志链当前点上做标记。这两种方式中，后者对应用系统有一定影响，随着标记点的频度提高，应用的Flush频度也越高；前者则具有一定的普适性，作用在文件系统处，至少能够保证文件系统这一层的一致性，但是仍然不能彻底保证应用一致性，只有后者才能做到真正端到端的一致性。用户进行回溯的时候，可以选择回溯到这些列出的一致点处。

保证应用层一致性就像在观看视频时所执行的回溯动作，回溯点处所对应的图像帧所表达的意义，可能是杂乱无章的。比如某人的手正处于半空中想要做些什么，但是观众单从这一帧图像来判断的话可能根本无法预测这个人要干什么，此时这一帧图像对于观众来讲就是不一致的。如果将视频回溯到这个点，让视频中的事件从这个点开始继续演化，那么此时系统就无法判断出这个人的手为何此时处于半空中，他是想抬起来还是想放下去呢？更不知道这个人此时具体想要做什么。但是如果将回溯点向未来再推进一些，那么此时对应的这帧图像可能恰好停留在此人抬起手握住一扇门的把手，那么此时这个事件就具有标志性意义，也就相当于数据系统中此时的数据状态是一致的，业务系统应用程序可以根据这个状态继续演变下去，比如产生多种情节：

- (1) 此人推开门走了进去，看到某某场景；
- (2) 此人突然想到了什么，犹豫了一下，又把手放了下来走了；
- (3) 僵住了，一直握着把手不放；
- (4) 其他各种发展路径，这也就对应了平行宇宙理论所设想的。

但是有人会有疑问：如果回溯到刚才那个不一致的历史时间点，难

道此时应用系统不能够针对这个人的手自己选择几种场景继续发展么？比如：①手继续向上抬起；②手向下放下；③僵在那不动了。其实，提出这个疑问的人自身已经回答了这个问题，也就是说，此时世界针对这个人的手的发展路径只给出了三种选择：向上、向下、不动。而第三种情况，也就是静止不动的场景下，其随后的发展要么也是向上或者向下，如果永远不动了，那么就证明系统出现问题了，这个人的路径已经无法再继续发展了。

所以，这个人的手最终还是要向上或者向下发展到一定的程度才可以体现出具体标志性意义，比如向上抬起握住把手或者捋了捋头发，抑或向下垂到了腿边。“抬起来、捋了捋头发”，这个动作包含两个子动作，第一个是“抬起手”，第二个是“放到头上”，第三个是“捋头发”。完成这一次对于观众有意义的动作，需要完成多个子动作才可以，而这其中任何一个子动作都是无法单独让观众看懂的，必须结合起来。而对于应用系统，也是这样的。每一笔业务逻辑（或称事务、交易，Transaction）是由多个原子操作（Atomic Operation）组成的，要完成一个有意义的业务逻辑，必须完成其包含的所有原子操作。而如果在一连串的原子操作之间出现问题，系统停机，那么这些原子操作在系统重新启动之后就必须进行前滚或者回滚从而保证一致性。

前滚就是将原本应该完成而未完成的原子操作继续执行完毕；而回滚则是指将自从上一次业务逻辑操作之后发生的原子操作全部回退，恢复到上一个业务逻辑完成时刻的状态，从而保证数据一致性。系统根据日志中所记录的所有操作以及状态来判断是需要前滚还是回滚。也就是说，电影里的那只手要么就向上发展到一定标志性时间点，比如抬起来握住把手，要么就向下回溯回之前的标志性时间点，也就是未抬起之前的状态，比如垂在两腿边。

那么，对于一个CDP系统，回溯时最好可以保证应用一致性以避免应用挂起这份CDP影像之后所经历的前滚或者回滚过程。而保证应用层一致性的方法，只能靠应用层解决，比如应用系统每提交一份Transaction之后，利用某种手段将这个事件通知CDP系统，后者从而可以做好相应的回溯标记。这也就对应着观众在看电影时，对关键时间点做标记，当回溯的时候，只要回溯到这些标志性时间点就可以得到一个可感知其意义的图像。

5. CDP的适用场合

作为真正的“数据时光机”，CDP以其叹为观止的技术革新，成功实现了人类梦寐以求的时光回溯技术，甚至可以改写数据历史（下文中有详细描述），当然只是在计算机数据存储领域。然而，这项技术的价值也是不菲的，企业部署起来也应综合考虑。

首先要考虑的**一定是RPO**。没有任何技术所提供的RPO可以媲美CDP了。从纯底层角度来看，CDP的RPO绝对等于0，即数据0丢失。这是理所当然的，每一个写IO都被同步到了CDP服务端，有什么理由RPO大于0呢？如果非要说大于0，也只能是上层不争气，就如上文中对于数据丢失的讨论一样。

然后还需要考虑**RTO**。值得肯定的是，目前的这种CDP解决方案架构，其所提供的RTO也是相当惊人的。不管是系统灾难还是个别文件等数据灾难，CDP所能提供的RTO都可以达到几分钟级别。CDP基于磁盘的数据镜像及Snapshot+CDP后处理，已经注定了它所能提供的RTO。恢复单个文件只需在主机端将对应时间点的影像挂载即可，主机无须重启。恢复整机操作系统也只需要将虚拟卷影像映射给相应主机并在主机端配置为SAN Boot即可。试问哪种方案还能够如此方便呢？

CDP架构也不是完美无缺的，首先它需要比受保护的数据源的存储容量更大的容量，相比多余的部分被用作IO仓库以保存持续的写IO数据。其次，由于写IO的持续镜像操作，势必会对主机性能造成一定的影响，如果CDP服务器使用的后端存储性能远低于数据源所用的存储性能的话，这种影响将会变得更显著。再次，CDP方案绝不适合大数据量改动的场合，比如经常有大文件被高速上传到受保护的卷，用不了多久便被删除，然后再上传新文件。这种情况下，CDP服务端很快便会被塞满，而且数据高速写入数据源，也就同样需要高速被同步到CDP服务端，如果是基于主机的IO捕获，则其对资源的耗费将不得不考虑进去。

CDP到底适合什么样的场景呢？总结如下：

- 对RTO和RPO要求甚是苛刻；
- 对源卷的数据更新IO尺寸较小但是每一笔都很重要；
- 强烈需要极细粒度的时光回溯或者历史改写，比如需要创建 N 个平行影像卷进行测试等情景；
- 关键业务宕机频繁；
- 不差钱。

符合上述几个要求的场合，部署CDP将会得到很高的投资回报率。

6. CDP与VTL

同样都是使用磁盘作为Online存储介质，当VTL遇见CDP，就像是小鸡遇见凤凰一般。VTL算个什么呀？就一虚拟磁带库，立着磁带库的牌坊却用硬盘来当存储介质，浑身都透着假。说它容量大，物理磁带库就笑了；说它耗电少，物理磁带库又笑了，磁盘一直转能不耗电么？说它备份和恢复速度快，连本地磁盘和NAS都笑啦！能快到哪去？备价格

式都和磁带一样，恢复之前还得啰啰嗦嗦扫描一大堆索引，创建一大堆文件列表。那它到底能干什么呢？大家都笑啦，异口同声地说：“给物理带库当个缓冲呗！”

VTL的处境正如它实际的作用一样，确实很尴尬。

你说都是同样的柜子同样的磁盘同样的接口，这同处一个机房的两个东西，它差别咋就这么大？买CDP吧，不差钱儿！

思考： 可以将所有级别的CDP理解为Database-Like。即任何操作均记录于日志中，日志可以前滚或者回滚，每条记录都有时间戳或者序号。本地CDP设备的容灾就可以考虑为类似Oracle Dataguard或者DB2 HADR的架构，其实本质上就是这样的。但是CDP的日志具体实现起来与数据库的日志还是有很大差别的。因为CDP可以瞬间提供任意时间点的直接可访问影像，如果使用数据库类日志记录方式，实现这个功能是绝对不可能的，因为数据库日志的恢复需要Replay过程，而这个过程将会耗费一定时间。CDP的日志是针对每个LBA或者LBA段

（Block）都保存一个日志链条，而不是全局的事件日志，这样可以保证RTO。此外，也不排除有些CDP设计为了简化操作和效率而直接使用数据库方式日志，恢复时，CDP根据给出的时间点，将时间点之前的所有日志进行Replay，但是又不能将日志中的数据操作直接覆盖了源卷（源卷需要保留以使得可以放弃当前影像），所以只能将日志Replay到另外的存储空间，而且必须以LBA为顺序来排放以便查找（日志中只包含被写过的块，对于没有被写过的块，依然存放于源卷中，所以Replay出去的块只是那些被写过的块，整个LUN影像是由源卷中未被写过的块和被Replay的块共同组成的，所以针对每个针对生成

影像的读IO，都需要先查找Replay出去的块），也就是将无序的日志处理成有序的LBA与数据的对应表。所以本质上来讲，CDP最终都要实现以LBA或者Block为单位的数据链条，如果使用类数据库日志的方式也只是拆东墙补西墙，看似写入日志的时候简单，但是最后恢复的时候却要将节约的时间再拿出来，而且还得贴上一部分存储空间。时间、空间、性能总是两两矛盾。所以说万物同源，虽处于不同层次却拥有相同的本质。

7. CDP与传统备份系统的比较

1) 传统备份所面临的挑战

传统的备份方法已经在企业中根深蒂固地使用了很长一段时期了。在主机端安装对应应用程序的备份代理，然后通过代理程序，将需要备份的数据传送给磁带机或者磁带库。备份任务一般每天都要执行，如果遇到数据量很大的情况，甚至在一次备份窗口所能够提供的时间内，比如晚10点到早6点，备份可能都并没有执行完，备份速度已经成为传统备份系统的一个主要瓶颈了。其次，在企业所制定的备份策略中，全备份一般都需要定期执行，比如一周执行一次，如果是大型企业，可能还需要保留数周甚至数月、数年之内的每次全备份磁带。在这种情况下，多次保留的全备份中，就会有相当一部分数据是重复的，如果打算节省空间，那么就得引入重复数据删除等数据缩减技术，这又是一笔额外开销。

另外，用磁带来备份大量小文件，此时将会是梦魇。备份数据时，备份软件都会预先生成待备份文件的索引以及其他的元数据，如果备份的目标是大量小文件，比如十万个小文件，那么光是索引阶段就要耗费

掉大量的时间，有的时候甚至可以持续数小时。

在需要做数据恢复的时候，传统的基于磁带的备份系统又会表现出多个问题。比如恢复速度慢，恢复之后不能立即可用，由于磁带介质或者磁带驱动器机械故障导致的恢复失败，恢复大量小文件速度极慢等，这些问题都是实实在在摆在眼前的，时常会让备份管理员痛心疾首。

2) VTL，加速传统备份

为了缓解磁带备份的弊端，VTL被发明了出来。VTL是为了解决备份和恢复速度慢的问题而生的。它利用磁盘来虚拟成一盘一盘的磁带，利用软件来虚拟一个或者多个厂商的多种磁带驱动器和机械臂，从而替代了传统的物理磁带库。由于使用了磁盘而不是磁带来作为存储介质，其表现出的速度也有了很大提升。当然单块磁盘的顺序读写速度有时甚至赶不上一盘LTO4磁带，但是如果将磁盘组成Raid之后，所表现出来的速度就远超过LTO磁带了。

然而，VTL并没有脱离备份软件成为一个独立的系统，它只是替代了传统的物理磁带库而已。底层使用硬盘作为存储介质，而上层却并没有表现出硬盘随机快速寻址的优势，对外还是表现为一种流式的顺序访问设备。其表层并未进化，这也是VTL的关键限制，除了速度加快、管理简单了之外，VTL相对于传统物理带库并没有本质上的进化，也并没有颠覆传统的备份系统。

并且，VTL虽然使用磁盘作为备份介质，但是它却并没有加速大量小文件情况下的备份，备份软件的索引过程依然需要执行。

3) CDP，彻底颠覆了传统备份架构

CDP意为Continuous Data Protection，即连续数据保护。CDP会记录主机对源数据所作的每一笔更改，并且将更改的内容记录下来。这样，就可以恢复到任何一个时间点时候的数据影像了。目前市场上的CDP产品，几乎都是相同的架构，即在主机端安装一个底层IO过滤驱动，将所有针对目标数据的写IO镜像写到CDP服务端设备上保存，CDP服务端会为每个Block保存一条日志链，每次针对这个块的写IO内容都被追加到链条末尾，并记录对应的元数据映射。

CDP保存了源数据的每一次改动，但是它与写镜像不同：写镜像情况下，针对一个块的后一次的写会覆盖之前的内容；而CDP却不会，针对每次写，CDP相当于都做了RoW操作。这样，就可以恢复到任何一个指定的时间的数据影像了。也就是说，CDP是一个高级的、可回溯的实时镜像系统。

这种把数据变化实时镜像保存的方法，彻底颠覆了传统的磁带备份系统。由于CDP的本质进化，被镜像之后的数据中几乎没有重复冗余数据。而且，更为神奇的是，在灾难发生之后，主机可以直接从CDP服务端来挂载任意时刻的卷影像，或者直接挂载系统盘对应的卷，直接从SAN网络启动主机，整个恢复过程非常快，远非传统备份可比。

由于CDP作用于块级别，所以与文件层面无关，纵使文件再小，数量再大，镜像和恢复的过程中也丝毫不会影响速度。

4) CDP将会是VTL的替代品么？

CDP完全脱离了各种备份软件，它只需要在主机端安装一个过滤驱动或者客户端管理界面程序。而VTL可以说是一个四不像的东西，并没有发挥出磁盘本来的优势。从这一点来讲，CDP完胜VTL。但是由于传

统备份的根深蒂固，以及CDP产品被认知和接受的程度还有待推进，还有成本等因素的制约，所以，短时间内CDP并不会完全替代VTL，或者说并不能彻底颠覆传统备份架构。但是长期来看，随着企业对数据安全以及RPO和RTO的要求越来越高，CDP将会逐渐打开并且占领备份市场的大部江山。

CDP产品基本上有两种形式：一个是纯软件，另一个是捆绑了CDP软件的硬件。不管是硬件还是软件，它们最终其实都是软件，将软件安装在某x86服务器上，后端使用各种扩展卡连接若干JBOD，便成了CDP硬件。

5) CDP相比传统备份所具有的优点

根据目前的CDP模型，数据从源到目的的是一个实时同步镜像过程，所以这种模式如果称其为备份的话，那么就是一种最彻底最纯粹的备份。它相比传统的备份有如下优点。

- 0备份窗口：数据实时的同步复制，源端的任何变化都同步地体现在备份端。再加上备份端的CDP和Snapshot功能，一个一致的备份瞬间就可以生成。
- 接近0的恢复窗口：可以直接从备份端将某时刻的卷挂载，直接使用。这个过程操作非常简单，耗费时间也相当于点几下鼠标。
- 具备实时容灾功能：传统备份系统只能将数据备份下来之后再传输到远程；CDP系统则可以直接充当实时容灾系统。
- 面对大量小文件的备份具有天生优势：传统备份系统在备份大量小文件的时候性能很差，耗费时间过多；而CDP则是在块级别运作，而且是实时同步，与文件的种类、大小等都没有关系。
- 回溯粒度能够达到秒级甚至更低，当然为了充分保证一致性，通

常恢复时都是选择一个一致点来恢复。

- 部署方便：主机端代理无须复杂配置，无须与各种应用相耦合。可以实现不停机部署。
- 数据重复率低：未变化的块都是共享的，而传统备份中，相同的块会被保存多份。
- 大量小文件的情况下，CDP有着根本性的优势。

6) CDP到底适合什么样的场景呢？

CDP虽好，但也不是完全适合任何环境的。比如，如果源数据端的写入流量非常大，比如200MB每秒，那么此时使用CDP就显得不合时宜了，会对主机端的IO性能产生一定影响，并且CDP服务端也会吃不消。

总体来讲，具有下列特点的环境，可以考虑部署CDP：

- 对RTO和RPO要求甚是苛刻；
- 对源卷的数据更新IO尺寸较小但是每一笔都很重要；
- 强烈需要极细粒度的时光回溯或者历史改写，比如需要创建N个平行影像卷进行测试等情景；
- 关键业务宕机频繁；
- 预算充足；
- 源数据端写入流量不是很大，或者大量小文件的环境下。

当然，与传统备份相比，CDP系统在成本方面也将是投入巨大的。究竟选择哪一个，还要看预算和需求的平衡结果。

8. CDP产品简介（EMC RecoverPoint）

目前CDP市场上的主打产品有两个，一个是存储行业巨头EMC经过

收购之后推出的RecoverPoint，另一个则是顶级存储软件提供商飞康（Falconstor）自主研发的IPstor CDP。两个的CDP解决方案，其架构如出一辙，在使用方式上也类似。前文中所提供的几个CDP架构模型均是这两家的CDP产品架构中提炼抽象而来，所以读者看过前文之后，就相当于已经了解了实际产品的架构。这里只对EMC的RecoverPoint产品做一个简要的介绍。

如图16-46所示，左右分别为RecoverPoint CDP服务器硬件的前视图和后视图。

图16-46 RP服务器

这个硬件其实是一台Dell R610 1u服务器，配有两个千兆以太网口和4个4Gb/s的FC接口（一张4口QLogic FC HBA）。操作系统为Linux Based。CDP功能实现于其上的核心应用程序软件。

如图16-47所示的是RecoverPoint的回溯粒度与普通的日备份和Snapshot的对比。前文中曾经提到过，由于CDP在数据一致性方面不如Snapshot可控性好，可以说根本不可控，这确实是个问题。但是RecoverPoint则提供了一种机制，即让用户自己记录对应的时间点都干了什么，回溯的时候就会有参考。甚至通过安装App Aware Agent的方式，还可以自动获取被监视的应用程序目前所处的状态，比如，Database Agent，一旦监控到数据库此时做了Checkpoint，则会通知CDP服务端将当前的时间点做一个Bookmark，以便在回溯恢复的时候让用户知道这个时间点的影像对数据库来说是一致的。RecoverPoint还可以监视其他多种应用，比如Windows下的VSS模块、MS SQL Server VDI等。

图16-47 CDP与Snapshot

如图16-48所示的窗口中就显示了这些显式的重要时间点以及对应的事件。

图16-48 一致性标记

另外，RecoverPoint不仅可以实现本地CDP数据保护，还可以实现跨越广域网实现异地CDP数据保护。其本质原理实际上是在本地和远程各部署一台CDP服务器，本地的CDP此时除了将IO镜像到自身的存储空间然后进行Snapshot+CDP后处理之外（或者自身没有存储空间，只作为IO转发器），还作为一个IO转发器，将收到的IO数据通过广域网同步到远程站点的RecoverPoint服务器，由远程CDP服务器来处理镜像的数据。这样，一旦本地站点整体灾难，远程站点依然可以保存一份可用的数据。远程Disaster Recovery功能在飞康IPstor CDP产品中同样提供。其整体架构如图16-49所示。

图16-49 整体架构

9. CDP产品配置实例

上文中介绍了EMC的Recover Point产品。本节中我们改用飞康IPstor CDP来为大家做一个CDP具体配置演示。

IPstor CDP同样也支持基于主机端的IO镜像和基于Fabric API的FC网络层IO镜像，这里我们只给出基于Windows操作系统的主机端IO镜像的演示。IPstor CDP运行于主机端的代理程序叫做“DiskSafe”，这个代理程序模块不但操控IO镜像，还负责创建Snapshot和挂载Snapshot。

(1) 如图16-50所示，打开DiskSafe主界面，左边的项目很简单，分别为Disk、Group、Snapshot、Event。其中Group中定义的就是任意多个卷的组合，形成一致性组。首先右击Disk，从弹出的快捷菜单中单击Protect命令。

图16-50 主界面

(2) 出现如图16-51所示的窗口，窗口中列出所有这台主机上的磁盘以及分区。可以针对整盘或者某分区进行镜像保护。在此我们选中Disk0准备对其进行整盘保护，单击Next按钮继续。

图16-51 选择要镜像的盘

(3) 如图16-52所示，代理程序便会与CDP服务器端通信以便查询是否服务器端存在已经创建好的镜像存储空间。本例中，新出现的窗口为空，所以我们需要在CDP服务端新建一个镜像存储空间，单击New Disk按钮。

图16-52 在服务器上创建镜像盘

(4) 如图16-53所示，出现Allocate Disk对话框。上半部分的窗口显示了目前所连接的CDP服务器端的名称和地址，下半部分则显示了镜像卷的大小。本例中大小与源卷相同，并且这个镜像卷挂载到主机的方式是使用iSCSI，由于本机没有安装FC卡，所以FC方式不可用。ThinProvision默认启用，即镜像卷占用的空间会随着实际源卷的IO不断同步逐渐增加，而不是一开始就占用所设定的空间，空间渐增分配单位设定为1024MB。

图16-53 创建新盘

(5) 如图16-54所示，单击OK按钮之后，CDP服务器端便开始创建这个镜像卷。创建完成之后便会出现如图16-55所示的窗口，此时，新卷已经出现在了窗口中。选中它后，单击Next按钮继续。

图16-54 创建过程

图16-55 选择新建的盘

(6) 出现如图16-56所示的窗口。在这个窗口中，提供了一些可以调节的参数，比如实时IO镜像模式（Continuous mode），即每个对源卷的写IO都同步传输到CDP服务端镜像卷之后才允许后续的源卷写IO；周期性镜像同步模式（Periodic mode），即本地先将一段时间之内（比如几秒钟）的源卷所有写IO的数据复制暂存在本地，然后一批一批地镜像到CDP服务端，相当于异步镜像。前者对本地IO性能有较大影响，而后者则影响较小，但是存在数据丢失的风险。我们用默认的参数即可。另外值得一提的是，代理端不但可以进行全卷扇区对扇区的完整镜像，而且还可以识别源卷中的NTFS文件系统，而只对文件实际占用的那部分空间数据进行镜像操作，这将会大大降低不必要的传输，降低初次同步时间。单击Next按钮继续。

图16-56 参数选择

(7) 如图16-57所示，这个窗口中可以设定对镜像卷做Snapshot的自动触发时刻表，每当一个Snapshot被触发，代理模块便会通知OS层面将FS缓存写入源卷，或者调用VSS服务对相关的应用进行Freeze。之后代理通知CDP服务端对对应镜像卷生成一份Snapshot，而这个Snapshot总是一致的。

图16-57 选择Snapshot时刻表

(8) 还可以对Snapshot做一些高级设定，如图16-58所示。

图16-58 时刻表高级设定

(9) 当完成Snapshot Schedule设定之后，出现Summary窗口，单击Finish按钮，如图16-59所示。

图16-59 完成

(10) 此时，在代理主窗口中的Disk项目右边就会显示出这个受保护的Disk0，其状态显示为Protection Initializing，正在执行同步镜像初始化操作。稍等片刻，状态就会变为Synchronizing状态，即首先将源卷上已经存在的数据同步到镜像卷。当初次同步完成之后，便进入了持续同步期，即源卷不断接收的写IO被不断同步到镜像卷。这个过程如图16-60所示。

(1)

(2)

(3)

图16-60 同步过程

(11) 镜像卷在主机端表现为一个物理卷，此时千万不要在操作系统自带的卷管理工具中对这个卷进行任何操作，因为此时是由代理模块在管理这个卷，如图16-61所示。

图16-61 镜像卷的状态

（12）在代理主界面中，按照图16-62所示的方法来手动创建一个Snapshot。

图16-62 创建Snapshot

（13）当需要从Snapshot中恢复文件的时候，则在左边窗口定位到Snapshot项目下，找到对应的Disk0，在右边的窗口中会出现系统针对Disk0所做的所有Snapshot。选中需要恢复的时间点Snapshot，右击，从弹出的快捷菜单中选择Mount Snapshot命令，如图16-63所示。

图16-63 挂载Snapshot

（14）Mount Snapshot之后，打开磁盘管理器，此时会发现这份Snapshot对应的虚拟影像已经被挂载到了主机上，代理会自动为其分配一个盘符（本例中为F盘），如图16-64所示。

图16-64 快照影像卷

此时便可以浏览这个影像磁盘中的内容，将需要恢复的文件复制出来了。

（15）如果需要整卷恢复，则更加方便，直接在对应的Snapshot上右击，从弹出的快捷菜单中单击Restore命令，如图16-65所示。

图16-65 Restore快照

（16）在出现的窗口中，单击Next按钮，如图16-66所示。

图16-66 单击“下一步”

(17) 在出现的窗口中，选择Disk or Partition单选按钮，单击Next按钮，如图16-67所示。

图16-67 选择分区恢复

(18) 如图16-68所示，在出现的窗口中，我们选择恢复一个Snapshot，选中对应的Snapshot，然后单击Next按钮。出现如图16-69所示的窗口。这里我们选择将Snapshot直接覆盖恢复到源卷，即Original primary disk，然后单击Next按钮。

图16-68 选择Snapshot

图16-69 选择覆盖方式

(19) 在图16-70所示的Summary窗口中单击Finish按钮，此时会出现恢复进度窗口，如图16-71所示。整盘Restore并且Restore到源卷的过程，与直接挂载Snapshot影像不同，前者是实体数据的恢复，即从位于CDP服务端的IO仓库中，参考对应的Snapshot与时间戳信息，将这份Snapshot中的所有数据覆盖到源卷。

图16-70 完成

图16-71 恢复过程

疑问：源卷中有部分数据可能并不需要被覆盖，为何不能直接重定向指针直接指向Snapshot入口呢？无能为力，因为，我们现在是对源卷进行恢复，而Snapshot和CDP都是针对源卷的一份镜像卷来生成的，这份镜像卷是在CDP服务器上的，CDP服务端并不维护任何源卷的地址指针，所以此时只能是将目标恢复时间点之前所有被CoW的数据块覆盖到源卷才可以。图16-72所示的是恢复结束时的状态。

图16-72 恢复完成

10. CDP优化设计

1) 日志链空间压缩

CDP作为一种数据连续保护设计思想，其功能无疑是强大的，但是其效率和空间占用也是让人头疼的问题。尤其是空间占用，保存每个写IO，可想而知其所耗费的空间总量有多大。如果能够采用一种用时间来换空间的方法来达到节能降耗的话，也不失为一种好办法。在这方面，我国华中科技大学的一项发明专利（公开号：CN101430657A）便非常巧妙地实现了这个目的，而且其算法并不复杂。下面对这个专利进行演绎，以便让大家在理解这个专利的基础上对CDP本身达到更加深刻的理解。

专利： 这项专利的核心思想和出发点是“读写IO具有局部性”。怎么理解？比如某个应用程序，拿记事本来当作例子，记事本打开某文本文件，文件中保存的是a~z这26个字符，总容量也没有超过一个扇区的大小。当打开文件时，文件系统会从磁盘上将这个文件实体内容对应的整个扇区读入缓存，并且复制到记事本程序的缓存中。之后，你用记事本将其中的头几个字符，比如a、b、c、d更改为1、2、3、4这四个字符，并且保存。此时，文件系统会将这整个扇区的数据再写回覆盖到磁盘上对应的扇区，而此时这个扇区所变化的内容只有4字节，其他均未变。这就是“读写局部性原理”，即应用程序在读或者写某份数据的时候，其实只是想要改变其中的5%~20%的数据。

如果按照CDP的做法，将会保存着整个写IO而不能判断只保存这个扇区中的前4字节。这只是用一个扇区来举例，实际中各种文件系统几

乎都是以簇为单位进行读写的，一个簇比如4KB、8KB甚至16KB或者更高，如果仅仅改变了几个字节就要将整个簇覆盖到原有扇区的话，那其中的无用功就太多了。当然，文件系统这么做也是必需的，我们不可能去改变文件系统，况且一般文件系统每次覆盖都不会去CoW出被覆盖的簇进行保存，所以文件系统的这种行为在对空间的浪费的影像上还是比较轻的，并不具有累积性质。但是对于CDP来说就不同了，每次都保存了大量不必要的内容，具有累积性，实在是浪费。好在我们可以在镜像卷上来实现任何想法。

得想一种办法，让CDP保存每个写IO的时候只保存变化的数据，或者用某种办法来实现类似的思想。但是，用传统扫描的方法去比对每个IO相对于前一个IO的变化量而只保存变化的部分，这样做是不太好了，因为效率太低，会影像性能。有一种既简单又成熟的算法能够快速地对两份数据中的异同，这就是XOR算法， $1 \text{ XOR } 1 = 0$ ， $1 \text{ XOR } 0 = 1$ ， $0 \text{ XOR } 0 = 0$ 。即相同的数据互相XOR之后的结果为0，不同的数据则为1。如果将两份扇区中的内容进行XOR运算之后，按照“读写局部性原理”，理想情况下（结尾会举出一个很不理想的情况），结果中会有80%~95%的连续0，而剩余的部分则会是不规则的0和1排列。此时，再使用压缩工具对这些连续的0进行压缩，效率将会大大提高。

疑问： 有人会质疑，既然两份扇区中的内容大部分相同，那么为何不直接使用压缩算法对其进行压缩呢？难道压缩连续的0比压缩乱序排列但是局部相同的1和0更为高效么？XOR是否是多余的一步？非也。压缩算法的局限性是很大的，举例来讲吧，随便找一份视频文件，复制一份，然后用某压缩工具对这两份文件打包压缩，压缩完毕后你会发现，其结果等于单独压缩每份文件的结果之和。

而最理想的情况应该是压缩之后的结果仅为一份文件的空间。既然两份文件内容完全相同，为何得到的压缩率却远大于50%呢？究其原因就是因为压缩算法的局限性，即一般压缩算法会设定一个滑动窗口，每次只读取待压缩目标文件的一定长度的数据，在这份数据中进行重复数据的搜索和压缩，将结果保存，然后将窗口往前移动，再读取接下来的定长数据，再压缩，然后结果追加到先前结果之后，最终达到文件末尾，压缩完成。为了保持一定的效率，窗口不可能太大，比如先扫描整个文件以找出重复的数据，甚至扫描所有给出的文件，找出全局重复的部分然后压缩，如果按照上两者的思想，则其压缩比将会显著提高，但是随之而来的是所耗费的时间以及系统资源也将会显著提高，这两个矛盾平衡之后的结果是，人们宁愿牺牲一些空间以换取效率的提高。本书其他章节中会介绍一种在全局数据空间内消除重复数据的方法。

在论述XOR之后为何压缩的效率和压缩比会更高的原因之前，我们先来论述一下专利CDP对IO的处理过程。

2) 专利中的CDP引擎处理写IO的过程

(1) 假设T1时刻CDP镜像卷某地址对应的Block内容为“abcabcabcabcabcabcabc”，即8个abc字串串联。这份数据用Bt1表示。T1时刻系统示意图如图16-73所示。

图16-73 T1时刻

(2) T2时刻某写IO欲将数据覆盖为“xyzxyzabcabcabcabcabc”，即头两个abc被更改为xyz。设这份新Block为Bt2。CDP截获这个写IO，然后用新IO与CoW出来的T1时刻的数据内容Bt1作XOR运算， $Bt2 \wedge Bt1 = Pt2 = "1111110000000000000000000000"$ （示意结果，并非二进制结果，下

同)。此时，CDP引擎首先将T1时刻的数据内容Bt1写入一个叫做“校验值链条”结构的首部，然后再将XOR的结果Pt2追加到“校验值链条”中Bt1之后，并将Bt2写到镜像卷中，即覆盖Bt1。这样，在时光回溯过程中，利用公式 $Pt2 \wedge Bt1 = Bt2$ ，便可反运算出Bt2的实际内容。T2时刻系统示意图如图16-74所示。

图16-74 T2时刻

(3) T3时刻，另一个写IO即Bt3到来，内容为“xyzxyzabcabcabcdefdef”，此时CDP引擎将镜像卷中对应地址的数据Bt2进行CoW出来，然后进行XOR运算， $Bt3 \wedge Bt2 = Pt3 = "00000000000000000000111111"$ ，将这个结果追加到“校验值链条”中最后一个结果也就是Pt2之后，同时将Bt3写入镜像卷。这样，在回溯时利用公式 $Pt3 \wedge (Pt2 \wedge Bt1) = Bt3$ 就可以反运算出Bt3的实际数据内容。T3时刻系统示意图如图16-75所示。

图16-75 T3时刻

随后的写IO依此类推。T6时刻的系统示意图如图16-76所示。

图16-76 T6时刻

然而，虽然可以用一份基准Block，即Bt1，随后的写IO轮流永远相互XOR下去，但是这样做容易导致XOR链条过长，在数据恢复反运算的时候，需要XOR的次数也就相应过多了，所以XOR链条过长就会影响回溯时候的性能。因此在下一步中，流程有了变化。

(4) 假设，在T7时刻，CDP截获了写IO数据Bt7，此时，CDP引擎依然是先CoW出Bt6，然后进行运算 $Bt7 \wedge Bt6 = Pt7$ ，依然将Pt7追加到Pt6

后面。然后，关键的一步，CDP引擎此时把Bt7追加到Pt7之后，同时将Bt7写入镜像卷覆盖Bt6。这一步结束之后，系统的示意图如图16-77所示。

图16-77 T7时刻

从第（1）步到第（3）步，是一个周期轮回，第（4）步则是新一个周期的开始。镜像卷上的每个扇区地址都会对应着这样一个数据链条，链条中的每一个周期中的第一个Block总是为对应时间点的写IO数据实体内容的复制，后来进入的写IO将与前一份写IO进行XOR操作并将结果再追加到链条中，循环执行。每个周期的时间可调，并且可以根据系统的负载等情况做到动态可调，比如负载很小，则可以适当将周期变长，负载很大，则周期适当缩短。

3）专利中的CDP引擎在进行时光回溯时的处理流程

（1）假设，用户输入回溯时间点T3，并且要求将T3时刻的源卷影像以虚拟卷的方式挂载到主机端。CDP引擎收到这个事件之后，便开始准备相关资源和相关进程，并且立即向主机端客户端映射出一个虚拟卷，其大小与源卷相同。此映射过程中不涉及对源卷中任何地址的映射操作。

（2）主机端开始对这个虚拟卷发起读IO操作，假设主机端需要读出Bt3这个Block，CDP引擎接收到这个IO请求之后，立即比对T1和T7时刻哪个时间距离T3时刻近，本例中T1时刻被选中，则引擎立即从对应Bt3这个Block地址所对应的校验值数据链中根据时间戳定位并读出Bt1、Pt2、Pt3这三个相邻的Block，按照正向恢复公式 $Bt3 = Pt3 \wedge (Pt2 \wedge Bt1)$ 来计算出Bt3的实际数据，然后立即发送给主机客户

端。同理，引擎接收到针对其他任何地址（比如Ct3、Dt3等）的Block的读请求后，都会根据相同的步骤读取对应地址的Block所对应的数据链中的相关的Block和Parity Block进行XOR反运算，并且将结果发送给主机。

（3）假设，用户输入回溯时间点为T5，并且要求以虚拟影像方式直接挂载此时此刻的虚拟卷。CDP接收到这个事件后，做与第①步相同的工作。之后，主机端发起针对Bt5的读请求，CDP引擎接收到这个IO之后，判断出T5时刻距离T7时刻比较近，所以CDP引擎根据时间戳定位并读出Bt7、Pt7、Pt6、Pt5这4个相邻的Block，按照反向恢复公式 $Bt5 = Pt6 \wedge (Pt7 \wedge Bt7)$ 计算出Bt5的实际数据并发送给主机。

（4）（本步在专利中并未做说明，由作者个人推演）接上一步，假设主机端要求对Bt5所对应的LBA地址进行写入操作。这时候，机制就稍微有点复杂了。主机可能同时在对源卷和虚拟卷影像进行读写操作，所以此时，CDP引擎接收到针对虚拟卷的写IO之后，一定要将其重定向到针对这个IO地址的一条新数据链中，并做好地址索引以便随时备查。如果再次接收到主机针对虚拟影像卷这个地址的写IO，则可以直接将新IO覆盖或者追加到新数据链中。

理解：为何要说“覆盖或者追加”呢？在此，有一个更灵活的技术实现，就是可以针对这份虚拟影像卷（注意，是虚拟影像，即经过一次CDP回溯之后的虚拟卷）创建二级Snapshot和CDP。如何做到的呢？很简单，上文说了，将写IO重定向到一个新数据链中，那么我们依然可以在这条新数据链中实现与前文相同的功能，只不过此时的基准卷变为原来镜像卷某时刻的影像而已。针对新基准卷的读IO都要经过时光回溯步骤，写IO则与一级CDP有一点不同，即无须CoW，而是用了RoW方式。

(5) (本步在专利中并未做说明, 由作者个人推演) 如果主机端选择以实体内容Restore或者叫做Rollback的方式来将源卷上的实体数据恢复到之前某一时间点的话, 则这个过程的底层机制与前文虚拟影像挂载模式下的实现方式没什么太大区别, 除了下面三点之外:

- 一是, CDP引擎不会映射给主机一个虚拟卷了;
- 二是, 然后将计算出来的回溯结果直接覆盖到源卷和镜像卷的对应地址;
- 三是, 删掉对应Block地址校验值数据链中这个时间点之后的所有Block, 仅保留此时间点之前的链条。

经过这些步骤之后, 源卷和镜像卷以及数据链条的整体状态就被Restore (或者叫Rollback) 到所给出的时间点了, 主机此时可以继续对源卷进行IO, CDP引擎从这个时间点上继续开始工作。此时, 原来所保存的晚于这个时间点的所有数据均被抹除了。这是Restore不如虚拟影像挂载的一点, 但是实体Restore之后, 相对于虚拟影像卷来讲, 却可以保证系统性能, 因为不需要复杂的回溯流程了。

4) 对校验值数据链的压缩问题探讨

现在我们要来探讨一下这个专利的初衷, 即, 使用XOR运算来节省大量校验值数据链所占用的空间的做法是否真的有效果。

参考CDP引擎IO处理过程中的第①步所给出的示例数据和第②步所给出的XOR结果, 假设, 压缩算法的窗口很小, 仅为一个Block的长度, 则每个滑动窗口仅对一个Block进行压缩, 数据Pt2的压缩结果可粗略地认为是类似“(6) 1 (18) 0”这种表现形式, 即6个1和18个0组成的字串 (这里我们忽略压缩算法的真实结果表示方式以及纯二进制结果,

大片连续的1的情况很少出现)。同理,针对数据块Pt3的压缩结果是“(18) 0 (6) 1”。

我们来对比一下,如果CDP引擎对每个新IO块不进行与前一个IO块的XOR运算而直接将其以实体内容的形式追加到数据链条末尾,这种情况下,调用压缩算法对每个实体数据块进行压缩,针对数据Bt2的压缩结果为“(2) xyz (6) abc”。可以判断,如果都转化为二进制的话,这个结果的长度将远大于“(18) 0 (6) 1”。

另外,我们以上只是假设了压缩算法的窗口大小仅有一个Block的长度,如果增加窗口大小,比如两个Block的长度,那么刚才的例子中就可以同时对Pt2和Pt3进行压缩,结果为“(6) 1 (18) 0 (6) 1”,而直接压缩实体数据块的结果为“(2) xyz (6) abc (2) xyz (4) abc (2) def”,可以看出此时压缩的比率相对直接压缩实体数据块所提升的更大。

所以,结论很明显,XOR运算之后的大片连续分布的0,显著地提高了压缩的效率和比率。

总结: 这个专利使用了非常普遍的XOR和压缩运算从而达到节省空间的目的(根据专利描述,可以节省20~30倍的空间占用,无疑是一个很有诱惑力的数值),这使得其实现起来的成本大大降低,而且,如果使用硬XOR运算和压缩解压缩芯片的话,便可以显著提升恢复效率,不失为一种很好的解决方案。但是也有一个小小的遗憾,即如果新写IO数据块相对于前一个数据块之间的相同部分的内容在两个数据块中是被错开排列的,哪怕错开1B或者1b,比如“abcdefg”和“0abcdef”,这种情况下,XOR后的结果将会是0和1的不规则不连续排列,压缩比

率大大降低，XOR运算恐怕只能额外增加系统负担了。

5) 日志链时间点合成

针对CDP数据占用空间过大并且管理开销过大的问题，还有另外一种妥协的方法，即CDP时间点合成法。CDP引擎为每个Block都保存一个按照时间先后排序的影像链，每个Block每发生一次写IO，这个IO就会被追加到这个Block链的尾部。如果用户觉得CDP的粒度太细了，没有必要，决定释放一部分空间，只保留几个关键时间点的影像，那么此时就可以将Block链条进行合成操作。

这里所谓的“合成”其实并不是合成，而是删除所有Block链条中给出的所有需要保留的时间点之间的数据块。比如，如果用户只要求保留T5和T8这两个时间点时LUN的影像，那么就可以将所有Block日志链中的T6、T7时间点对应的Block留存删除掉，这样就节省了大量空间。

6) 全局日志链

如果在CDP系统设计中做出一部分妥协，损失一部分灵活性，就可以大大降低CDP的复杂度，获得更高的处理效率。前文中所设计的CDP系统，允许实时地Mount某个时间点的虚拟影像，这样虽然非常灵活方便，但是这么做就需要为每个Block维护一个日志链，计算开销和存储开销都很大。如果对于某个LUN维护一个全局Block日志链，即针对这个LUN的所有写IO都保存在同一条日志链中，不再区分Block，那么维护开销将大大降低，但是同时，也不再可能实时地输出某个时间点的LUN影像了。正如前文中的“思考”框中描述的一样，如果只维护一个全局日志链，那么在用户需要访问某个时间点的LUN影像的时候，系统就只能将日志链从对应的时间点开始向回Replay到一个空LUN中，然后结

合源卷中未被写过的块，共同组成一个影像。这样做效率也是比较低下的。

一种取代的做法是：只需要对每个LUN整体维护一个单一的Block日志链即可，用户选择回溯到哪个点，就将整个Block日志链从头开始到这个点之间的日志Replay到一个空LUN中，但是被Replay出去的Block在空LUN中要被按照与LBA地址一一对应的方式写入。与此同时，将源卷中未被写过的Block也按照LBA地址一一读出并且写入空LUN。这两个过程都完成之后，这个LUN中的内容就是对应时间点的一份真实影像而不是虚拟影像了。在第一次Replay之后，如果用户再次选择回溯或者前滚到另一个时间点，那么可以从Block日志链中上一次Replay结束的地方开始向用户给出的那个时间点在日志中所处的位置进行Replay（快进或者快退），将块覆盖到影像LUN中，二次Replay所耗费的时间就仅仅是Replay日志链的时间了。

另外，全局日志链模式由于将所有的Block变更都混合存放在同一个链条中，所以每条日志记录都需要记录对应的Block被覆盖之前的内容以及覆盖之后的内容。为何要这样做呢？因为在Replay日志时，在回退操作的时候，必须知道对应的块被覆盖之前的内容（虽然可以通过查找的方式来查找日志链中这个块上一次被修改之后的内容，也就是本次修改之前的内容，但是这样做是非常慢的）。所以，全局日志链中会有将近一半的相同内容（针对同一个块的两条日志记录中会有一份内容相同），占用额外空间比较大。

将全局日志链与分块元数据链相结合使用可以获得最佳的效果，正如上文中“雕心镂月幻影斩”中所表述的。

7) 日志写缓冲处理

由于CDP模块需要针对每一个写来做RoW/CoW操作，CoW操作会使得写操作的延迟大大增加，而RoW操作由于要写入随机分布的块日志链，属于随机度非常大的写入操作，所以延迟也会非常大。在前文所述的CDP普遍使用的模型中，如果CDP服务端阵列处理写的速度由于RoW/CoW的影响而与主机写入主阵列的速度相差太大的话，那么势必严重影响主机端性能。基于此，CDP服务器可以先将所有进入的写IO写到一个日志中缓存，从而达到快速响应写IO请求的目的，系统在后台系统不繁忙的时候重放这份日志然后进行CDP处理。写日志的过程是完全连续IO，所以此时可以大大降低IO延迟。

16.2.6 VSS公共快照服务

VSS的全称是Volume Shadow copy Service，中译名即“卷影复制服务”。乍一看挺抽象，而且也容易被其中译名弄晕，所以我们擅自称它为“公共快照服务”。上文中曾经说过，为了保证Snapshot的一致性，几乎所有存储厂商都提供了自己开发的针对各种应用程序和文件系统的代理模块。而应用程序有无限多种，存储厂商也有多个，但是这些应用以及存储代理都运行在同一个操作系统中，与其每一个厂商为每一种应用程序都开发自己的代理，不如在操作系统中建立一个公共的Framework服务，往上适配各种应用程序，往下则适配各厂商的代理，做到统一控制调配，统一开发接口。微软在其Windows Server操作系统中就提供了这样一种公共服务模块，这就是VSS模块被开发的初衷。

VSS的架构如图16-78所示，左边是具体的VSS架构，右边是抽象后的架构。

图16-78 VSS架构图

整个VSS逻辑架构包含4个部分：VSS核心服务、Writer、Provider、Requestor。

- **Writer**。这里的意思就是代表运行于操作系统中的各种应用程序，当然这些应用程序必须支持VSS服务，运行时便向VSS进行注册以表明自己的身份。每一个应用程序就是一个Writer。至于为何叫做Writer，是因为这些应用程序都需要向对应的卷或者磁盘中写入数据，所以叫做Writer，即数据写入者。在Windows系统中典型的Writer比如Exchange Server、SQL Server、Oracle、DB2、Share Point等。VSS核心服务提供SDK开发接口，任何应用程序都可以借助它来支持VSS。
- **Provider**。这里的意思代表各个底层存储系统中的快照支持者/提供者，说白了就是各个存储厂商的存储系统以及其在主机端的代理程序，存储系统必须支持Snapshot功能，不支持Snapshot则不属于Provider。
- **Requestor**。这里的意思是代表各个存储厂商的快照管理程序，当然也可以是快照代理程序。这个程序负责何时触发快照，管理员通过在这个程序中设定一些Schedule来控制Snapshot的生成时间点和频率等。

使用VSS来作为公共快照服务之后，生成一个快照的具体流程如下。

(1) T1时刻，某应用程序，比如Exchange Server，正在运行，并且持续不断地向底层卷LUN1写入数据。

(2) T2时刻，系统管理员需要针对LUN1触发一次快照以保存当前的数据影像，管理员打开对应存储厂商的Requestor程序界面，在其中手

动触发Snapshot。

(3) T3时刻, Requestor程序接收到了管理员的操作, 然后立即向VSS核心服务请求将目前所有已经在VSS服务中注册的Writer, 也就是应用程序的列表返回给自己。

(4) Requestor判断Exchange Server是否在列表中, 如果在列表中, 则提取其注册信息, 并且用Exchange Server的注册信息以及所要进行Snapshot的卷列表向VSS发起请求, 通知VSS快照请求已经发起, 请协助处理后续事宜。

(5) T4时刻, VSS收到Requestor的请求之后, 根据Requestor传送的卷列表信息向所有Provider (Snapshot代理程序) 发起查询以得知哪个Provider可以提供针对LUN1的快照操作, 对应的Provider收到查询请求后会进行应答。

(6) T5时刻, VSS此时已经得知了所有必要的信息, 即哪个应用, 哪个卷, 哪个Provider。VSS立即向Writer, 即Exchange Server发起请求, 让Exchange Server暂时生成一致点, 没完成的运算赶快完成, 来不及完成的就暂挂, 将缓存中的数据该回退的回退, 并且暂停对数据卷的写IO操作。Exchange Server完成这些步骤之后, 会通知VSS它已经准备好了。

(7) VSS接收到了Writer的通知之后, 立即向对应的Provider发起执行快照的请求。Provider (代理程序) 收到VSS的指令后立即与存储设备通信, 通知存储设备立即对相应的卷生成一份快照。如果在VSS请求发起之后10秒钟之内没有收到Provider快照成功的通知, 则此次操作失败, 系统恢复原状。

(8) 一旦快照在存储设备上成功地生成，Provider（代理程序）就会接到存储设备的通知，然后Provider立即通知VSS快照已生成。VSS接到通知后立即向Writer发送通知告诉Writer可以继续进行针对相应数据卷的写IO操作。然后，VSS通知Requestor本次快照成功生成。

(9) 系统恢复常态。

VSS为不同的应用和不同的快照代理提供了一个公共Framework，极大简化了系统的复杂性。如图16-79所示为VSS介入前后系统API复杂度的变化。

图16-79 复杂度的变化

16.2.7 快照、克隆、CDP与平行宇宙

克隆的出现恰恰为平行宇宙、蝴蝶效应和弦、膜等理论提供了计算机建模运算研究的数据基础。

如果把我们现在对计算机所做的事情对应到造物主对现实世界所做的事情，那么就可以找到对应，如图16-80所示。那么是不是也就意味着现实世界的历史也是可以回滚的，也是可以重演的，只是我们现在的破解进程还没有到那一步。搞不好会让现实世界崩溃掉而回到原点？

图16-80 克隆与平行宇宙

思考： 同一个微粒为何会同一时刻出现在不同位置？我们通过某些实验隐约推测到这种结论，那么这个微粒是不是被多个快照（平行宇宙）所共享？是不是在同一个时间维度下附带有多个空间维度呢？弦理论就是这样一种理论，它似乎正试图去

寻找造物主到底为某个世界创建了多少个并行发展的克隆。而这些所有克隆的源头，也就对应了宇宙大爆炸理论的那个所谓的中心点。

强子对撞机是否会破解上帝代码的最终手段？是否可以探知到上帝构造世界的基石是1和0还是“有”和“无”，或者就是某种可以看到的東西，或者根本什么都没有？或者当对撞的一瞬间，上帝构造的世界被我们弄崩溃了？霍金为何先后抛出“不要与外星人接触”以及“时光回溯是可以实现的”两个理论？这些现在谁也不知道。

16.2.8 高帧率IO级数据录像

如果说快照是数据照相机，那么CDP就是数据录像机。目前多数CDP产品所实现的都是本书前文中所描述的那种低帧率录像，也就是每隔几秒钟对数据日志进行采样，然后对每个采样点根据回溯线进行制表操作（生成一张对采样点的时刻数据地址映射描述表），然后删掉位于采样点之间的数据以腾出空间。

低帧率的数据录像可以从一定程度上满足需求，但是毕竟用户的要求是越来越多、越来越膨胀的，24帧的电影其实足够了，但是现在依然出现了48帧的高流畅度电影。标清电影也能凑合看，但是就有人希望看到主角脸上的毛孔。对于数据也是这样，比如用户就需要恢复某个时间点的数据，结果恰好就这个时间点没有采样。

数据库系统的CDP方式

“细粒度数据回滚和前滚”这个思想到底是谁发明或者提出来的我们已经无从考证了，但是最成熟的高帧率数据录像机，还得是数据库系统

莫属。数据库系统对每一笔操作都记录得非常详细，包括这笔操作所做的更改类型（插入、删除、更新等）以及该笔操作所对应的SCN序号等。记录了所有变更操作，就可以任意回滚和前滚，当然，需要将一段时间内所有发生的变化都记录下来，才可以在这段时间内的历史数据中任意回滚/前滚。

数据库所不能实现的

数据库虽然可以任意回滚，但是它不能并行生成多个历史时刻的、可访问的数据库影像。如同一份十分钟之前的影像，一份一小时之前的影像，想同时访问这两个影像，此时数据库系统是无能为力的。

存储系统对**CDP**的特殊要求

作为一个存储系统，必须实现上述功能才算专业。但是如何在保证细粒度回溯点的情况下，依然还能提供上述功能，就是个比较大的挑战了。低帧率模式下，每个采样点会用一份表来记录映射关系，当挂起这个采样点成为一个虚拟的卷影像之后，针对该影像的IO访问，可以通过查询该表迅速得到执行。

但是高帧率模式下，采样点模型已经不适用了，由于回溯粒度非常高，甚至达到了每个IO级别，那么此时不可能为每个IO点都记录一张表。所以此时必须使用数据库日志形式的回滚方式，但又要同时实现并行影像访问，这似乎不可能。

IO级回放的唯一产品——飞康CDP

我们在前面虚拟化章节中提到了飞康，在CDP领域，不知道飞康就

别说你懂CDP。飞康是业界第一家、也是唯一一家能够提供真正可行的IO级回放粒度CDP技术的厂商。飞康进入中国时，引来国内多家厂商纷纷研究和效仿，不仅仅是在技术层面，更多是在产品、用户体验层面，因为几乎没有其他厂商能将CDP从技术到产品转化地如此彻底。业界也曾经质疑过飞康是否真的能够做到IO粒度的回滚，本书前文中也曾对IO级别的回放进行过分析以及对一致性方面的担忧。但是飞康很好地处理、包装和屏蔽了这些问题，让IO级CDP变得真正可用，这就是其差异化技术所在了。

说明： 由于飞康并没有公开这些技术的底层细节，所以外界对飞康的研究都属于猜测分析，那么我也不妨在这里分析一把，正如本书前文中的模型建立和分析一样。

试想一下，在这种IO级别细粒度回溯的场景下，就完全没有必要为每个IO记录一个单独的表了，也就是说，日志本身，已经是个内嵌的表了。我们先来做个简易模型，有100个LBA组成一个卷，然后针对这个卷的10、20.....100这10个LBA做了多次乱序、重叠的写IO操作，现在要求生成第二个IO（LBA70也就是T1时刻）和第9个IO（LBA10也就是T2时刻）历史时刻的两份影像，而且要求可并行访问。如图16-81所示。

图16-81 IO级CDP日志空间

我们还是按照之前的模型方法，假设这两个历史时刻虚拟卷已经生成，主机端已经挂载这两个卷，然后主机针对T1时刻的虚拟卷的LBA50进行了读操作，此时CDP系统应该去哪里找LBA50？读者应该很清楚了，那就是要从源卷的基准镜像中读出LBA50然后发送给主机。为什么？因为T1这个历史时刻点之前，对LBA50的更新还没有发生（虽然在

当前日志中T1时刻后面确实有一笔LBA50的写IO被记录），也就是在日志中T1时刻之前未出现过LBA50。同理，主机针对比如LBA1024（不在日志中）的访问，同样也要去源卷基准镜像中寻找。但是针对LBA10的访问，CDP引擎就必须从日志中读出LBA10返回给主机，因为在T1时刻之前，日志中恰好有一笔针对LBA10的写IO被记录。再来看针对T2时刻的虚拟卷的访问，其道理是一样的，同样是访问LBA50，但是此时CDP引擎就必须明确地知道，LBA50是存储在日志中的并且是在T2时刻之前发生，所以必须从日志中来找到T2时刻之前最近的一笔（可能会有多笔）LBA50的记录从而返回给主机。

这个模型非常简单，任何一个人通过看图都能很快判断出任何一个IO访问，到底应该从哪里找数据返回给主机。仔细体会一下，寻找其中规律，这个算法模型其实说白了就是：针对任何历史时刻点的虚拟卷影像的任何LBA的访问，CDP引擎首先要判断出该IO地址对应的数据是否存在于日志中，如果有，再去日志中该历史时刻点之前的记录中去搜索，如果搜不到，就从基准镜像中读出并返回；如果初始判断该访问地址根本就不在日志中，则直接从源卷基准镜像中对应的LBA中读出该数据即可，不需要搜索，因为该LBA与访问的LBA是一一对应的。

我们看到，上述算法模型，对针对某地址的访问，做了两级搜索，为何不用一级搜索呢，比如任何一个访问，先整体搜一遍日志，搜不到再去基准镜像中读取？如果这样的话，性能会非常差。上图中的日志只记录了十几笔操作，但是现实中，会有几十万、上百万甚至千万笔操作，搜一遍谈何容易。那么如何快速地判断某地址是否在日志中或者不在日志中？这个就很简单了，利用一个Bitmap，源卷每个块或者LBA，对应Bitmap中的一个bit，如果该bit被置1，证明该块/LBA目前在日志中存在，但是并不能判断存在于该历史时刻点之前还是之后；如果该bit被

置0，则证明该块/LBA在日志中不存在，也就是说从录像开始之后，这个块/LBA根本没有被更新过，此时皆大欢喜，直接去基准镜像中访问该块，性能最高。

第一级搜索可以利用位图来缓解性能问题。但是如果必须去日志中查找，此时性能一定会有损失的。但是我们还是可以进行优化。首先，为了加速查找，必须为整个日志生成一份元数据表，记录每个IO的序号和这个IO的起始目标地址和长度，比如以LBA扇区为粒度，使用32bit地址的话可以描述2TB的源卷容量，所以，实际产品可以选择一个支持原卷容量的上限，比如2TB。然后再使用32bit来表示IO的序号，也就是整个日志空间可保存约42亿次IO，每秒5K IOPS的话可以连续跑10天，一般保存10天的细粒度回放日志基本够用了。IO的长度使用SCSI协议中的上限（也就是16bit）足够了。

这样的话一共需要80bit来描述日志空间中的一笔IO，加上其他考虑（比如保留一些将来用的字段等），按照96bit也就是12字节来算，每笔IO记录将耗费12字节描述，每笔IO平均大小按照4KB来计算的话，元数据比例大概在3%左右，完全在可接受的范围内，如果日志空间大小为1TB，那么会耗费3GB内存空间来完全缓存元数据，这也是可行和可接受的。另外，这些元数据在没有生成虚拟影像的时候，是不需要常驻内存的，仅当在生成了虚拟影像之后，为了加速主机IO访问所以才有必要常驻内存。

元数据表只按照IO序号排序是不够的，因为IO序号不能被用作用户回滚时候的依据，因为没有任何人会对IO序号有概念，所以还必须在日志元数据表中加入时间戳，这里可以变通一下，可以对每笔IO都加上一个时间戳，也就是用时间戳来取代IO序号，如果不需要IO级别细粒度恢复的话，时间戳粒度可以粗一些，比如日志中每10个IO就记录一个时间

戳，这些都是可以灵活设计的。

元数据表是按照时间戳/序号来排序的，当用户生成了某个时间戳的虚拟卷影像之后，根据前文分析，每一笔针对虚拟影像的IO会经历两级搜索，如果落入了日志空间中，那么CDP引擎就要将该时间戳之前的所有日志搜索一遍以查找日志中所有与该IO请求的地址范围有交集的那些IO记录，然后将数据拼接起来，返回给主机。由于日志中的IO记录是按照时间/序号排序的，所以为了提升搜索速度，必须另外生成一份按照所记录IO的LBA起始地址排序的索引，也就是说，用户每生成一个某时间戳处的虚拟卷影像，CDP引擎就需要将该时间戳之前的日志部分的元数据做一个索引，这样就可以加速查找。比如接收到主机发来的读IO请求，系统在查完位图之后发现必须从日志中寻找该地址数据，那么系统会根据该IO请求起始地址到对应该时间点生成的索引中快速定位到所有覆盖了该IO起始地址的记录项（该索引按照IO起始地址排序，所以查起来很快），然后找出时间最新的那个记录项，按照该记录项中给出的其所处于日志空间中的位置，读出对应数据，返回给主机。

有了上述的模型和优化算法，基于IO粒度回滚的高帧率数据回放，就变得可行。如图16-82所示，还可以继续优化设计，比如将元数据空间单独拿出来，放到SSD中去加速访问。用户可以任意生成多份、多个历史时刻的虚拟卷影像，每个影像所占用的开销只是速查索引而已，上层用户体验就会非常给力。还可以将CDP日志先缓存在RAM中，然后定期Flush到磁盘上，相当于分成在线日志和归档日志两大部分，另外，归档日志中较早的数据如果不再需要回放则可以被Merge到源卷基准镜像中去，相当于一个合成备份，这样就不必维护越来越多的日志空间和元数据了。

图16-82 IO级CDP的四个存储空间

另外，在设计上还可以做更深层次的优化。长时间录像不是做不到，而是代价太高，比如需要大量的元数据被记录，这样重放时的性能会较差。有没有办法来降低元数据记录量从而进一步提升重放性能呢？飞康在底层其实是用了快照+录像合体来解决这个问题的。有技术感觉的读者一定会猜测到，其实就是以照片为基础，然后记录在每份照片基准之上的数据变化，这样所保存的元数据就非常小。这个原理，与实际的视频压缩编码非常类似，实际视频压缩编码技术中也是将视频分成P帧和I帧，P帧是基准，第一个I帧只记录相对于P帧的增量，第二个I帧则记录相对于前一个I帧的增量，利用 $P + I_1 + I_2 + I_3 + \dots + I_n$ 的方式，对整个视频进行微分，那些变化量不大的场景（比如某个镜头），几秒之内视角和内容基本相同，这样，这个镜头就可以被编码成 $P + nI$ 的形式，如果镜头切换，内容几乎全变，那么就使用新的 $P + nI$ 来记录，多个 $P + nI$ 积分就可以还原出整个视频。飞康CDP底层恰恰就是这种原理。

飞康会在CDP日志中打入很多TimeMark，每个TimeMark其实就是一个P基准帧，也就是一份快照（一份照片）。系统维护快照所耗费的元数据相比于CDP是要少得多的，因为快照不会记录IO的顺序和时间，只会记录位置映射关系，所以代价比CDP小得多。基于每个快照，系统会在快照的基准上，记录增量日志及其元数据，这样，对T1时刻的查找就是基于T1时刻之前距离T1时刻最近的Snapshot TimeMark进行的，基于它，与后续记录的CDP元数据共同组合，抽取出T1时刻的卷虚拟影像（TimeView）给主机使用。如果T2时刻与T1时刻之间没有Snapshot TimeMark触发，那么T2时刻与T1时刻的虚拟影像提取就得使用同一个Snapshot Marker，如果它们之间有Snapshot Marker，T2时刻便会以离它最近的Snapshot Marker为基准，与录像区新增记录的元数据共同抽取出T2时刻的TimeView。

上述的技术逻辑看似复杂，但是被封装成产品之后，只需要几步简单配置即可完成。但是如果你不了解技术原理和架构，这几步配置中的选项就会感觉非常迷茫。我们下面就来看一下飞康CDP的配置步骤，掌握了飞康的步骤，也就等于掌握和理解了其他CDP厂商产品的原理和配置步骤了。

（1）如图16-83所示，要对某个源卷进行录像，就必须像按下录像键一样，在界面中勾选Enable CDP，完成这个向导之后，系统后台就会自动开始录像了。

图16-83 Enable CDP日志记录及选择存储空间

（2）然后设置日志空间的管理参数，包括连续录像周期、是否自动扩容、自动扩容阈值、扩容比例、日志容量上限。在“高级”按钮中还可以设置日志从被Merge到源卷基准镜像空间时的IO力度，过高的力度会影响性能，因为会占用较多的磁盘性能资源。如图16-84所示。数据被从日志空间Merge到基准镜像中之后，日志中已被Merge的数据并不会立即被删除，两种情况发生时才会被删除：CDP日志空间不够用了，被强行删除；CDP日志超过了保留时间。

图16-84 设置日志空间管理参数

（3）再往下是设置快照参数。飞康对快照的处理与CDP不同，快照耗费资源比CDP要少。图16-85中所示的TimeMark相当于快照的时间戳，在这个页面中可以选择何时、开始每隔多长时间做一次快照以及最大保留快照份数、在快照生成时是否要保障该快照的一致性。

图16-85 设置快照策略

飞康在快照技术方面的实现是非常全面的，在主机端提供了支持各种主流应用系统的快照Agent组件，并且可以实现从CDP设备侧主动向主机上的Agent发出快照通知，这一点是其他产品或者方案中没有见过的，类似方案基本都是通过一个单独的快照管理软件来制定和执行策略，这显然带来了不便。毕竟我们都希望在一个页面中搞定一切最好。

（4）这一步结束之后系统便会在后台持续录像，并且在所指定的TimeMark时间点生成对应的快照。此外飞康很人性化地提供了手动创建快照、对快照以及CDP日志当前时间点增加说明性文字，以及设定快照优先级的功能，当快照数量达到上限之后，系统会优先删除低优先级的快照，如图16-86所示。另外，飞康CDP还提供了克隆功能，从任何一份快照生成一份实体卷数据，可以算作一个备份。

图16-86 对快照点做标记

那么之前系统记录的录像或者照片该怎么回放呢？回放功能在飞康产品中的名称对应为TimeView，也就是可以任意查看该数据卷在任意历史时间点的历史数据。可以从快照中生成卷虚拟影像，也可以从CDP日志中的任何时间点生成虚拟影像，当然时间总要有个最小粒度可供用户来选择，飞康提供了微秒级别的回放粒度。本书前文中也说过，专业性不强的用户是无法理解“IO序号”的，所以给出时间粒度才是最有意义的。

如图16-87所示，可以使用滑动条来任意寻找时间点，可以手动输入精确到毫秒级的历史时间点，也可以直接选择之前所标注的助记标记点，还可以单击放大按钮来精确查看每个时间点的详细信息。CDP回放的最佳用户体验全部集中在这个页面入口中了。图像的纵坐标表示该时

刻虚拟影像所占用的额外空间的容量，高度较高则表示该时刻的写IO压力比较大。单击放大按钮之后，会出现如图16-88所示的窗口。

图16-87 从CDP日志创建虚拟卷影像

图16-88 可视化的细粒度回放点

这个窗口会显示以5分钟为单位的回放点，左下角切换按钮可以切换到上一个或下一个5分钟。可以看到，纵坐标可以精确到每个时间点系统接收了多少个写IO。可以单击这个页面，会有竖直方向的参考线，页面下方会给出当前参考线所处的时间点以及该时刻的写IO压力，也就是该时间点占用了多少额外的空间。用户可以选择任意时间点进行回放。

提示： 这里要理解一点，IO压力为0的那些时间点并不意味着不能被选择用来生成虚拟卷影像，相反，这些点是用来生成一致性虚拟影像的最合适时机。

说实话，飞康这种对用户体验的雕琢真的是达到了很高的水准，

图16-89 基于IO级别的回放

我之前曾经在某厂商设计了一套所谓“可视化智能存储”的存储软件解决方案，飞康的这种设计可谓是英雄所见略同，有很强的共鸣产生。另外，如图16-88所示，当单击Search按钮之后，不但可以搜索该范围内的那些通过与快照Agent配合产生的一致性时间点，还可以选择过滤搜索那些低于所给出值写IO数量的时间点，因为这些小压力时间点往往具有更好的一致性，可恢复的几率更高，如图16-89所示。

此外，除了使用TimeView来回放历史时刻之外，还可以直接像数据库系统一样，回滚和前滚源数据卷，可以任意回滚或者前滚，回滚之

后如果不满意还可以前滚，直到满意为止。

虽然说CDP相当于数据录像机，但是真的把它做成和电影播放器一样的用户体验，飞康是第一个也是唯一一个。

说明： 飞康技术团队对技术的追求、表达和展现已经不能够用专业来描述了，应该用“偏执”来描述，偏执的产品往往是最让人崇敬的产品，也是质量最高的产品，偏执意味着执着的追求，一个团队能够允许偏执的存在，这个团队才会有顽强的生命力。

以上详细介绍了飞康的CDP技术。下面简单介绍一下飞康的CDP产品。软件定义存储的核心就是强大的软件层，至于硬件，目前已经完全倒向开放式x86 Server模式了，那些定制化、差异化的控制器，似乎已经没什么吸引力了。如果将飞康的软件平台IPStor Gen2 Platform安装在一台高性能服务器上，再加上磁盘扩展柜，那就是一套完整的存储系统，飞康CDP管理器就是这样一套偏重于数据保护CDP以及备份的存储系统。

如图16-90所示为飞康网关型
CDP管理器及一体化CDP管理器。网

图16-90 飞康CDP网关型管理器

关型产品只提供一个控制器，其后端可以挂接来自其他存储系统的逻辑卷；一体化产品则可以提供磁盘扩展单元来作为本地存储空间，另外也可以挂接其他系统的逻辑卷。飞康CDP管理器是一款软硬件打包的产品，用户可以直接购买软件License然后自行安装在自己的服务器上，但是对于一些非专业用户，则更加倾向于购买整套软硬件系统，一来免去安装烦恼，二来还可以保证更好的兼容性。

如图16-91和16-92所示为飞康IPStor Gen2 Platform以及CDP产品的特色技术和高级功能。这些技术和功能我们已经在本书前面的虚拟化章节中进行了详细介绍，在此不再赘述。

图16-91 飞康CDP网关型管理器的特色技术一览

图16-92 飞康CDP网关型管理器的高级功能一览

16.3 数据备份系统的基本要件

- 备份对象：是指需要对其进行备份的备份源，比如一台服务器上某块磁盘上的所有数据，或者某数据库下所有数据文件，这些都算备份对象。
- 备份目标：是指将备份对象的数据备份到何处。备份目标可以是备份对象本身的磁盘、磁带等介质，也可以是任何其他地点的磁盘、磁带机、磁带库等介质。如果备份目标位于备份对象本身，比如，从一台Windows服务器的D盘复制某些文件到自身的E盘，则不需要占用任何网络资源，因为数据从备份对象自身生成，到自身结束。如果备份目标位于其他地点，如同一个机房内的其他服务器，或者外部独立磁盘阵列，则数据从备份对象生成，传输到备份目标的过程中，就需要占用网络资源，因为连接备份对象和备份目标的只有网络。稍后将会详细介绍用哪种网络。
- 备份通路：也就是我们上文提到的，如果备份对象和备份目标都位于同一个角色上，那么备份通路就是这个角色自身的计算机总线，也就是连接这个角色的CPU、内存和磁盘的总线，因为数据只在这个总线上流动。如果备份对象和备份目标处于远离状态，则二者必须通过某种网络连接起来，而这个网络就是这个备份环境的备份通路。关于备份通路会在后面的例子中详细描述。
- 备份执行引擎：有了备份源、备份目标和连接二者的备份通路之后，需要一个引擎来推动数据从备份源流到备份目标。这个引擎一般由备份软件来担任。
- 备份策略：是指备份引擎的工作规则。引擎不能无时无刻地运转，它需要根据设定好的规则来运转。

下面将着重介绍一下备份目标、备份通路和备份引擎这三个要件。

16.3.1 备份目标

1. 用本地磁盘作为备份目标

用本地磁盘作为备份目标，就是把本地磁盘上待备份的数据，备份到本地磁盘其他的分区或者目录。用这种方式可以不影响任何其他服务器以及共用网络。数据流动的范围完全局限在备份对象自身。

但是这样做的缺点就是对备份对象自身的性能影响太大，数据从磁盘读出，需要耗费磁盘资源。读出后写入内存，需要耗费内存资源，然后再从内存写入磁盘的其他分区，同样需要耗费磁盘资源。备份执行期间，还会对其他IO密集型的程序造成极大影响。通常这种方式只用于不太关键的应用和非IO密集型应用，以及对实时性要求不高的应用。E-mail服务器的备份就是典型的例子，因为E-mail转发实时性要求不高，转发速度慢一些，对用户造成的影响也不会很大。

2. 用SAN磁盘作为备份目标

用SAN上的磁盘做备份，就是把备份对象上需要备份的数据，从本地磁盘读入内存，然后从内存中写入连接到SAN的适配器，即HBA卡缓冲区，HBA卡再通过线缆，将数据通过SAN网络传送到磁盘阵列上。

这种方式的优点就是从本地磁盘读出数据，写入的时候只耗费SAN共用网络带宽资源，而且能获得SAN的高速度，对备份对象性能影响相对较小；缺点是对公共网络资源和盘阵出口带宽有一定影响，因为耗费了一定的带宽用来传输数据，同时数据在流向盘阵接口的时候，也要占

用接口带宽。

如果备份对象数据本身就存放在SAN上的磁盘，而备份目标同样是SAN磁盘，那么数据流动的通路比较长。在后面介绍备份通路的时候再做详细阐释。

3. 用NAS目录作为备份目标

用NAS目录作为备份目标，就是将本地磁盘上的数据备份到一个远程计算机的共享目录中。比如Windows环境下常用的文件夹共享，就是这样一个典型的例子。一台计算机共享一个目录，另一台计算机向这个目录中写入数据。而数据一般是通过以太网来进行传递的。这种方式占用了前端网络的带宽，但是相对廉价，因为不需要部署SAN。

4. 用SAN上的磁带库作为备份目标

用FC接口作为外部传输接口的设备，不仅仅有主机上的HBA适配器、磁盘阵列，磁带机和磁带库，也可以用FC接口作为外部传输接口。用线缆连接磁带库和SAN交换机之后，处于SAN上的所有主机系统便会识别出这台磁带库设备，自然也就可以用磁带来当作备份目标了。

磁带库，由机械手、驱动器、磁带槽组成。图16-93和图16-94分别为某型号磁带库的外视图和某型号磁带库的内视图。

图16-93 某型号磁带库的外视图

图16-94 某型号磁带库的内视图

磁带驱动器是磁带库的核心组件。可以将驱动器想象成一个电机，带动磁带旋转，然后磁头贴住磁带、读写数据，把电机、磁头以及控制

电路集合到一起，形成一个独立的模块，就是驱动器。

机械手，在机械生产线上机械手就是一个计算机夹子而已，把物品从一个地方移动到另一个地方，当然这还需要程序来控制。而且像图16-94中所示的一样，机械手不一定是那种铁臂抓手，而只要它能寻找磁带槽上的磁带并将其推入磁带驱动器，就可以称其为机械手。

磁带库的工作流程如下。

（1）由机械手臂从磁带槽中夹取一盘磁带，推入磁带驱动器，驱动器完成倒带、读写等动作。

（2）读出完成后，退带，机械手臂夹取磁带，放回磁带槽，然后夹取另一盘磁带放入驱动器，重复刚才的动作。

整个流程都需要由程序来控制机械手臂和驱动器，那么程序运行在哪里呢？当然是连接磁带库的主机上，程序生成符合协议的电信号，经过HBA卡传送到磁带库电路板上，经过芯片处理，转换成操控机械手臂和驱动器的另一串电信号。所以连接磁带库的主机上，除了需要安装HBA适配器的驱动程序之外，还需要安装磁带库机械手和驱动器的驱动程序，这样才能够按照驱动程序定义的规则，来生成符合规定的、磁带库可以识别的电信号。

磁带机比磁带库功能少，但是基本原理都是一样的，只不过机械手臂没了，取磁带和放磁带需要用人手而已。而且一台磁带机同一时刻只能操作一盘磁带。而在磁带库中可能有多个驱动器，或者多个机械手，当然机械手不需要那么多，因为一个机械手就能完成，除非驱动器多得让一个机械手都忙不过来，这是不太可能发生的。多个驱动器可以同时读写多盘磁带（每个驱动器一盘），使得效率大大提升。

提示： 近年来，出现了一种虚拟磁带库产品，即用磁盘来模拟磁带。当然，磁盘就是磁盘，不可能变为磁带，那磁盘是怎么被虚拟成磁带的呢？当然是通过存储控制器来虚拟化。

话说回来，是磁带还是磁盘，这都取决于数据服务器看到的影像，而它看到的也不一定就是实际上存在的。虚拟磁带库也正是利用了这个原理。使用磁带库的是主机服务器，如果让主机服务器看到的影像就是一个磁带库，而实际上却是一台磁盘阵列，那么主机照样会像使用磁带库一样使用这台虚拟的磁带库。要做到这一点，就必须在磁盘阵列的控制器上做虚拟化操作，也就是要实现协议转换器类似的作用，一边以磁带库的逻辑工作，另一边以磁盘阵列的逻辑工作。

虚拟磁带库的好处如下。

- 速度大大提升。因为向磁盘写入数据要比磁带快。
- 避免了机械手这种复杂的机械装置，取而代之的是控制器上的电路板。
- 管理方便，随意增删虚拟磁带。

提示： LTO磁带的“鞋擦”效应。目前最新的LTO是LTO5代技术，它能够在不压缩的情况下每秒写入180MB的数据。但是磁带驱动器有个缺点，就是一旦开始写入，电机就会以全速运转，以额定速度写入数据，一旦备份软件提供给驱动器的数据速率达不到驱动器额定速度的时候，这种情况称为“欠载”，那么有些驱动器则可以降低转速，有些则保持转速而降低数据存储密度来适应欠载。但是驱动器有一个最低转速和最低密度，当备份软件连这个速度也满足不了的时候，磁带驱动器只能够先暂停下来，停止转动，然而又不能急刹车，急刹车会拉坏甚至拉

断带子，所以只能慢慢停下来，这就导致磁头所处位置比数据截止位置超前一段距离。当数据进入的速度能够满足最小速度或者密度之后，磁带驱动器再次尝试写入数据，但是此时磁带需要被倒带至上次的的数据截止位，然后继续将积攒在缓存中的数据写入，当速度再次不匹配时，执行相同的过程，周而复始。这就是鞋擦效应。鞋擦效应会严重影响磁带的寿命，每次磁带的同一位置被磁头划过，这个位置的寿命都会有所降低。解决鞋擦效应的办法就是引入D2D2T备份模型，比如VTL等，用磁盘来作为磁带的大缓冲，或者提高前端数据生成的速率。

5. 信息生命周期管理

假如，某企业有磁盘阵列容量共1TB，磁带库容量10TB。每天均会新生成5GB左右的视频文件，这些视频文件都存放在磁盘阵列的一个500GB的卷中。这样不到3个月，这500GB的卷便会被全部用完。而这个企业频繁调出查看的视频，一般都是最近一个月以内的，如果将一个月之前的、几乎很少或者永远也不会被再次访问到的视频也放在磁盘阵列上，这无疑是一个巨大的浪费，因此完全可以把这些文件备份到磁带库，而腾出磁盘阵列上的存储空间，供其他应用程序存储数据。

磁盘阵列是高速数据存储设备，而磁带库是低速数据存储设备，所以为了各得其所、物尽其用，有人便开发了一套信息生命周期管理软件，这种软件根据用户设定的策略，将使用不频繁的数据，移动到低速、低成本的存储设备上。比如只给某个视频应用分配20GB的磁盘阵列的空间，但是向它报告500GB的存储空间，其中有480GB其实是在磁带库上的。

这样，应用程序源源不断地生成视频数据，而管理软件根据策略，

比如某视频文件超过了设定的存留期，便将它移动到磁带库上，腾出磁盘阵列上的空间。但是对于应用程序来说，总的可用空间还是在不断地减少。虽然磁盘阵列上可能总是有空间，但这些空间是给最近生成的文件使用的，因为这些文件会被频繁访问。如果一旦需要访问已经被移动到磁带库上的文件，则管理软件会从磁带库提取文件，并复制到磁盘阵列上，然后供应用程序访问。

6. 分级存储

基于信息生命周期管理的这个思想，目前很多厂家都在做相应的解决方案，分级存储就是这样一种方案。

- 第一级：一线磁盘阵列，是指存储应用频繁访问数据的磁盘阵列。其性能相对二线和三线设备来说应该是最高的。
- 第二级：二线虚拟磁带库。这个级别上的存储设备，专门存放那些近期不会被频繁访问的数据。其性能和成本应该比一线设备低，但是性能不能太低，以至于提取数据的时候造成应用长时间等待，虚拟磁带库，正好满足了这个要求。虚拟磁带库利用成本比较低廉的大容量SATA磁盘，性能适中的存储控制器，这样保证了性能不至于像磁带库一样低，成本又不会像一线设备一样高。
- 第三级：磁带库或者光盘库等。这个级别上的设备，专门存储那些几年甚至十几年都不被访问到的，但是必须保留的数据。磁带库正好满足了这个要求，这是毫无疑问的。

16.3.2 备份通路

1. 本地备份

本地备份的数据流向是：

本地磁盘 → 总线 → 磁盘控制器 → 总线 → 内存 → 总线 → 磁盘控制器 → 总线 → 本地磁盘。

即数据从本地磁盘出发，通过本地的总线和内存，经过CPU运算少量控制逻辑代码之后，最终流回本地磁盘。

2. 通过前端网络备份

通过前端网络备份的数据流向是：

本地磁盘 → 总线 → 磁盘控制器 → 总线 → 内存 → 总线 → 以太网卡 → 网线 → 以太网 → 网线 → 目标计算机的网卡 → 总线 → 内存 → 总线 → 目标计算机的磁盘。

即数据从本地磁盘发出，流经本地总线和内存，然后流到本地网卡，通过网络传送到目标计算机的磁盘上。

这里说的前端网络，指的是服务器接受客户端连接的网络，也就是所谓“服务网络”，因为这个网络是服务器和客户端连接的必经之路。

后端网络，是对客户封闭的，客户的连接不用经过这个网络，后端网络专用于服务器及其必需的后端部件之间的连接，比如，和存储设备，或者应用服务器和数据库服务器之间的连接，这些都不需要让客户终端知道。

后端网络可以是SAN，也可以是以太网，或者其他任何网络形式。以太网并不一定就特指前端网络，也可以用于后端。随着以太网速度的

不断提高，现在已经达到了10Gb/s的速率，所以以太网LAN作为后端网络，同样也是有竞争力的。但是说到SAN，一般就是特指后端网络。

3. 通过后端网络备份

通过后端网络备份的数据流向是：本地磁盘→总线→磁盘控制器→总线→内存→总线→后端网络适配器→线缆→后端网络交换设施→线缆→备份目标的后端网络适配器→总线→内存→备份目标的磁盘或者磁带。

提示：这里说的“后端网络适配器”，泛指任何形式的后端网络适配器，比如FC适配器、以太网卡等。

4. LAN Free备份

LAN Free这个词已经在存储领域流行使用多年。它的意思是备份的时候，数据不流经LAN，也就是不流经前端网络。由于历史原因，导致了人们的思维定势，认为LAN只用于前端网络，所以说到了LAN就想到了前端网络，然而，我们上文已经做了解释，后端网络同样可以使用以太网LAN。LAN这个词本意为Local area network，即局域网络，它没有对网络的类型加以限制，可以说存储区域网络也是一个LAN。

思考：笔者认为这个词不再适合当今存储领域，取而代之的应该是Frontend Free这个新的名词，即备份的时候，数据不需要流经前端网络，而只流经后端网络。

Frontend Free备份的好处是：不耗费前端网络的带宽，对客户终端接收服务器数据不会造成影响。相对于后端网络来说，前端网络一般为

慢速网络，资源非常珍贵，加上前端网络是客户端和服务端通信的必经之路，所以要尽量避免占用前端网络的资源，备份数据长时间频繁地流过前端网络，无疑会对生产造成影响。解决的办法就是通过后端网络进行备份，或者本地备份。

无论是本地备份，还是通过网络备份（前端网络或者后端网络），都需要待备份的服务器付出代价来执行备份，即服务器需要读取备份源数据到自身的内存，然后再从内存将数据写入备份目标，对主机CPU、内存都有资源耗费。是否能让服务器付出极小的代价，甚至无代价而完成备份任务呢？当然可以。

5. Server free备份

这个名词是指，备份的时候，数据甚至不用流经服务器的总线和内存，消耗极少，甚至不消耗主机资源。下面来分析一下。

要想使备份数据不流经服务器本身，那么首先备份对象，即待备份数据所在的地方肯定不能是服务器的本地磁盘，因为数据从磁盘读出，第一个要流经的地方就是总线，然后到服务器内存，这样就不叫Server free了。

所以，备份源不能在服务器上，同理，备份目标也不能在服务器上，不然写入的时候照样流经服务器的总线和内存。那么到底怎样才能实现Server free呢？

很简单，备份对象和备份目标都不在服务器上，不在本地就只能在SAN上了。做到这一点还不够，因为主机要从SAN上的一个磁盘取出数据，写入SAN上的另一个磁盘，同样需要先将数据读入到主机的内存，

然后再写入SAN。那么到底怎样才能做到Server free呢？

答案是，用SCSI的扩展复制命令，将这些命令发送给支持Server free的存储设备，然后这些设备就会提取自身的数据直接写入备份目标设备而不是发送给主机。或者用另一台计算机作为专门移动数据之用，即待备份的主机向这台数据移动器发信号，告诉它移动某磁盘上的数据到另一个磁盘，然后这台数据移动器从SAN上的源磁盘读取数据到它自己的内存而不是待备份主机的内存，然后写入到SAN上的目标磁盘。

提示： 所谓的Server free，并不是真正的不需要服务器来移动数据，而是让服务器发出扩展复制命令，或者使用另一台专门用作数据移动的新服务器，来代替原来服务器移动备份数据，释放运算压力很大的生产服务器。当然，SAN上的源磁盘和目标磁盘或者磁带，数据移动服务器都需要有访问权。

为了统一数据备份系统中所有节点之间的消息流格式，Netapp公司和Legato公司合作开发了一种叫做NDMP的协议（网络数据管理协议）。这个协议用于规范备份服务器、备份对象、备份目标等备份系统各种节点的数据交互控制。服务器只要向支持NDMP协议的存储设备发送NDMP指令，即可让存储设备将其自己的数据直接备份到其他设备上，而根本不需要流经服务器主机。

16.3.3 备份引擎

备份引擎，就是一套策略、一套规则，它决定整个数据备份系统应该怎么运作，按照什么策略来备份，备份哪些内容，什么时候开始备份，备份时间有没有限制，磁带库中的磁带什么时候过期并可以重新抹掉使用等。就像引擎一样，开动之后，整个备份就按照程序有条不紊地

进行。

1. 备份服务器

那么备份引擎以一种什么形式来体现呢？毫无疑问，当然是运行在主机上的程序来执行，所以需要有这么一台计算机来做这个引擎的执行者，这台计算机就叫做“备份服务器”，意思就是这台计算机专门管理整个数据备份系统的正常运作，制定各种备份策略。

思考： 备份服务器的备份策略和规则，怎样传送给整个数据备份系统中的各个待备份的服务器呢？和汽车一样，车轮和引擎之间有传动轴连接，备份服务器和待备份的服务器之间也有网络来连接，那么通过以太网还是通过SAN网络来连接呢？

答案是以太网络，因为以太网络使用广泛，以太网之上的TCP/IP编程已经非常成熟，非常适合节点间通信。相对于以太网，SAN更加适合传送大量数据。而利用前端网络连接还是利用后端网络连接呢？

一般我们常用前端网络来连接待备份服务器和备份服务器。因为备份策略就好比两个人之间说了几句话，所以把这几句话传送给待备份服务器，不会耗费很大的网络资源，充其量每秒几十个包而已，这对前端网络影响非常小。有了网络连接，我们就有了物理层的保障。

但备份服务器是如何与每个待备份的服务器建立通话的呢？它们之间怎么通话？通话的规则怎么定呢？这就需要待备份服务器上运行一个程序，专门解释备份服务器发来的命令，然后根据命令，做出动作。

这个运行在各个待备份服务器上的程序，就叫做备份代理（Backup Agent），它们监听某个Socket端口，接收备份服务器发来的命令。比

如，某时刻备份服务器通过以太网前端网络，给某个待备份服务器发送一条命令，这条命令被运行在该待备份服务器上的备份代理程序接收，内容是：立即将位于该服务器上C盘下的XX目录复制到E盘下XXX目录。备份代理接收到这个命令之后，就会将该待备份服务器上C盘下XX目录复制到E盘下的XXX目录。如果D盘是本地盘，E盘是一个SAN上的虚拟磁盘，那么实际数据流动的路径就是：

本地磁盘 → 总线 → 内存 → 总线 → SAN网络适配器 → 线缆 → SAN交换设施（如果有） → 磁盘阵列。

数据源源不断地从本地磁盘流向SAN网络上的磁盘阵列，成功备份之后，备份代理收到来自待备份服务器操作系统的成功提示，然后备份代理通过以太网向备份服务器返回一条成功完成的提示。这样备份服务器便会知道这个备份已经成功完成，并记录下开始时间、结束时间、是否成功等信息。

同样的道理，如果例子中的E盘也是本地盘，那么路径就短多了。如果备份服务器告诉备份代理，将数据复制到位于SAN上的磁带库设备而不是磁盘阵列，那么同样，备份代理将数据从本地磁盘读出，然后通过SAN网络适配器发往磁带库。当然这需要在待备份的服务器上安装可以操控磁带库设备的驱动程序。同理，E盘如果是个NAS目录，则数据便会被发往远端的NAS服务器了。

2. 介质服务器

设想：假如在一个数据备份系统中，有一台普通的SCSI磁带机连接在某台主机上，且有多台主机的数据需要备份到这台SCSI磁带机的磁带中，而SCSI磁带机只能同时接到一台主机上，总不能搬着磁带机，给

每个机器轮流插上用吧？

当然可以，但是这样很麻烦。有没有一种办法来解决这个问题呢？当然有了。可以将这台磁带机连接到固定的一台计算机，只能由这台计算机来操作磁带机。其他有数据需要备份的计算机，和这台掌管SCSI磁带机使用权的计算机，通过以太网连接起来（当然也可以通过其他网络方式连接，但前面说过，以太网是最廉价、最广泛使用的网络），谁有数据，谁就将数据通过以太网发给这台掌管磁带机的计算机，收到数据后，这台计算机将数据写入只有它才有权控制的磁带机。写完后，下一台有数据需要备份的计算机，重复刚才的动作。

这样，我们用了一台计算机来掌管SCSI磁带机，然后在这台计算机的前端，我们用以太网扩展了连接。虽然微观上磁带机只有一个接口，只能连接一台计算机，但是经过以太网的扩展之后，这台磁带机成为了公用设备，掌管磁带机的计算机成为了代替这些服务器行使备份动作的角色，因为整个数据备份系统中，只有这台计算机掌管了备份目标，也就是磁带机、磁带，所以我们称这台服务器为“介质服务器”。也就是说，这台服务器是数据备份系统中备份介质的掌管者，其他人都不能直接访问备份介质。

思考： 这台计算机只是掌握了备份介质，谁都可以向它发起请求，然后传输需要备份的数据给它。但是如果同时有多台服务器向它发出请求，怎么办？

显然还需要一个调度员，来管理调度好多个待备份服务器之间的顺序，做到有条不紊，按照预订的策略来备份，避免冲突。那么谁来担当调度员呢？前面讲到的“备份服务器”本身就是这样一个调度，所以非他莫属了。

将这个调度员也接入以太网，调度员使出它的必胜大法——在每个待备份的服务器上，都安装它的“耳朵”和“嘴巴”，即备份代理程序，通过这个耳朵和嘴巴，调度员让每台服务器都乖乖地听话，按照顺序有条理地使用介质服务器提供的备份介质进行备份。在一个数据备份系统中，介质服务器可以有多台同时分担工作。

图16-95是一个Veritas Netbackup备份软件的备份流程。

图16-95 备份流程

（1）某时刻，备份服务器发起备份，它通知“介质服务器A”备份“待备份服务器C”上的相应内容。

（2）“介质服务器A”向服务器发出指令，告诉它可以进行备份了，请发送需要备份的数据。

（3）待备份服务器C把需要备份的数据通过以太网发送给“介质服务器A”。

（4）“介质服务器A”将收到的数据源源不断地写入磁带机。

（5）重复第（1）步，只不过介质服务器为B，待备份服务器为A。

上面这个拓扑图，是一个Frontend unfree备份方式。因为备份数据流占用了前端网络带宽。

至此，我们的数据备份系统中，已经有了三个角色了：备份服务器（调度员）、介质服务器（仓库房间管理员）、待备份服务器（存储货物的人）。

再转回去看看还没有出现“介质服务器”前的那个例子。会发现那时候，仓库房间尚充足，仓库每个门都开着，每个人都可以从各自的门进去存放物品，每个需要存放货物的人都有权直接访问它们的仓库房间，它们只靠一个调度员来协调。这时候可以认为，每个消费者都在管理着仓库房间，它们每个人都是仓库房间管理员，只不过它们各自管理自己的房间而已。所以，这种情况下，每台待备份的服务器，都是介质服务器，而且每台介质服务器因为都需要操控备份设备，所以还需要安装诸如磁带库等设备的驱动程序。而备份服务器只有一个。

相对于由存放货物的人自行管理仓库房间的情况，由专人来管理仓库房间，所耗费的前端网络资源更大。因为存储货物的人，首先需要通过网络将货物发送给仓库管理员，然后管理员再将货物放入仓库。而如果让存储货物的人自己存放，则会省去第一步。

图16-96所示的就是用上述思想进行数据备份的一个拓扑图。

图16-96 Frontend Free

(1) 备份服务器通过以太网，同时向三台介质服务器（也是待备份服务器）发出备份开始指令。

(2) 待备份服务器直接将数据通过后端SAN网络设施写入备份目标。A和B的备份目标是磁盘阵列，它们可以同时写入备份目标。C的备份目标是磁带库，如果磁带库只有一个驱动器，则同一时刻只能用于一个备份操作，这个例子中，磁带库被C独占。当然C完成备份之后，磁带库可以被其他服务器使用。A、B、C都安装有磁带库机械手和驱动器的驱动程序。

上面的拓扑图（图16-96），就是一个典型的Frontend Free备份方

式，因为备份数据流不占用前端网络带宽，而只有备份时所发送的指令数据经过了前端网络。

随着各种应用系统的不断出现，比如，各种数据库管理系统、E-mail转发处理系统、ERP系统、办公自动化系统等，备份技术也随之飞快发展，传统的备份操作，仅仅是备份操作系统文件，即不管这个文件是何种类型，被什么应用程序生成和使用，备份的时候统统当作一个抽象的备份源来看待，只需要把这个文件整体传送到备份目标就可以了。

如今，用户的需求越来越高，越来越细化。比如，某用户要求只备份某数据库中的某个表空间，或者只备份某个E-mail。由于一个表空间包含一个或者多个数据文件，这样命令发给备份代理的时候，只能是这样：“备份某数据库下的某表空间”，而不是：“备份X盘X目录下XXX文件”。因为调度员不可能知道某个表空间到底包含哪些具体的文件。要完成这个备份动作，必须由运行在待备份服务器上的某种代理程序来参与。那么这个代理是否可以上文提到的备份代理呢？

完全可以，但是有个需要增加的功能，即这个代理程序必须可以与待备份的应用程序进行通信，从而获得相关信息。可以有两种方式来完成备份：直接获取到这个表空间对应的数据文件有哪些，然后自行备份这些文件；或者直接调用数据库管理系统的命令，向数据库管理系统发出命令，备份这个表空间。

如果是第一种方式，将调度员发出的命令对应成实际文件的工作。比如，需要备份的是一个DB2数据库上的某个表空间，那么这个代理程序需要与DB2实例程序进行通信，DB2实例服务程序告诉备份代理，XXX表空间对应的容器（数据文件）为某某路径下某某文件。备份代理获得这些信息后，直接将对应的文件备份到备份目标。

如果是第二种方式，备份代理收到调度员指令之后，便会向DB2实例服务程序发出指令：“db2 backup db testdb tablespace userspace1 online to \\.\tape0”，这样，就利用DB2自身的备份工具备份了数据。目前广泛使用的做法都是第二种方式，因为解铃还需系铃人，用应用程序本身的备份工具进行备份是最保险的方法。

综上所述，对于每一种待备份的应用程序，都需要一个可以和该应用程序进行通信的代理程序，这就需要开发针对各种应用程序的代理。目前，像Symantec Backup Exec备份软件，提供了诸如Oracle数据库代理、DB2数据库代理、Exchange代理、Lotus Notes代理、SQL Server代理、SAP代理等诸多应用程序的备份代理。

图16-97中的ServerFree Option就是用来实现ServerFree功能的一个模块。Agent for开头的选件，就是针对各种应用程序所开发的备份代理程序。NDMP选件，用来实现ServerFree所需要的协议栈。

图16-97 各种备份代理及插件

SAN Shared Storage Option，是用来管理那种“每台待备份服务器都是介质服务器”情况下，各个服务器对备份目标设备的共享使用权的一个模块。因为每台服务器都装有目标设备（磁带库等）的驱动程序，都可以控制磁带库，为了避免意外冲突，SAN Shared Storage Option作为一个附加的选件来协调各个服务器之间有顺序地使用目标设备而不发生冲突。

16.3.4 三种备份方式

1. 完全备份

假如某时刻某文件中只包含了一个字符A。此时我们对这个文件做了备份操作，将其复制到其他介质上，这份备份的文件中只包含字符A。

稍后，这个文件被修改，在字符A之后增加了一个字符B。此时我们又对这个文件做了备份，将其复制到其他介质上，这份备份的文件中包含字符A、B。

以后，不管这个文件怎么变化，变成多大，包含多少个字符，只要备份，就将这个文件整个备份下来，这就是完全备份。

2. 增量备份

假如某时刻某文件中包含100个数字1、2、3、.....、100，在对其做完全备份后，文件中增加了一个数字101，又过了一段时间，文件中又增加了一个数字102，而我们已经有了一个完全备份，这个备份已经包含了文件中1~100这100个数字，如果我们此时再次对这个文件做完全备份，是不是太浪费时间和存储空间了呢？得想出一种办法，即只备份自从上次完全备份以来发生变化的数据。

完全可以，我们以一种自己可以识别的格式，将101和102这两个数字保存到一个单独的文件中。这就叫做增量备份，意思就是只备份与上次完全备份内容之间相差的内容。如果要恢复这份最新的文件，只要将上次完全备份和最后一次的增量备份合并起来，便可组成最终的最新完全备份，从而恢复数据。

增量备份要求必须对数据做一次完全备份，从而作为增量的基准点。否则随意找一个基准点，所生成的数据是不完整的。

3. 增量备份

经过了完全备份和差量备份，我们已经有了两个备份文件：包含数字1~100的完全备份和包含数字101和102的差量备份。如果这份文件每天都会增加一个数字，而我们每天都要做差量备份甚至完全备份么？这里还有一种选择，就是增量备份。

增量备份是指：只备份自从上次备份以来的这份文件中变化过的数据。这里的“上次备份”，不管上一次备份是全备、差备，还是增备自身，本次增量备份只备份和上一次备份结束的时刻，这份文件变化过的数据。比如，我们现在拥有包含数字1~100的完全备份和包含数字101和102的差量备份。此时，我们打算以后每天执行增量备份，那么，第二天，这份文件增加了一个数字103，所以我们只备份103这个数字，依此类推，第三天我们只备份104这个数字，这样备份速度极大地加快了，备份所消耗的空间也小了。

提示： 在实际使用中可以灵活地制定各种策略，比如每周一对数据进行完全备份，周二到周五每天对数据进行增量备份等。

如果对数据进行增量或者差量备份，普通的文件，备份软件一般是可以检测到文件相对上次备份时候所发生的变化。但数据库的备份，备份软件想检测某个数据文件的变化，一般来说是不可能的，因为这些文件内部格式是非常复杂的，只有数据库管理软件自身才能分析并检测出来，所以每个数据库管理软件（如Oracle、DB2等）都有自己的备份工具，可以全备、差备和增备，而第三方备份软件在对数据库做备份的时候，只能调用数据库软件自身提供的各种命令，或者程序接口。

16.3.5 数据备份系统案例一

前面介绍了数据保护的基本原理和大体思路，下面来看下现实中的数据备份领域，了解一下现今广泛实行的数据备份都是怎么做的。

下面用一个企业的IT系统作为一个初始化的例子，如图16-98所示，某企业IT系统现有FTP服务器一台，E-mail服务器一台，基于SAP的ERP服务器一台，DB2数据库服务器一台（用于SAP服务器的后台数据库），备份服务器一台，大型FC磁盘阵列一台，小型磁带库一台（一个机械手，两个驱动器）。

图16-98 某公司IT系统示意图

以上五台服务器各用以太网卡连接到同一个以太网交换机上，同时各个办公室的客户端PC也通过局域网连接到这台交换机上。另外，除备份服务器之外的四台服务器上分别装有一块FC HBA适配卡，且通过光纤连接到一台SAN交换机上。磁盘阵列也用一条光纤连接到SAN交换机上。该企业使用Symantec Backup Exec 11D备份软件进行备份操作。

在备份服务器上需要安装Symantec Backup Exec的软件Master Server模块。

在每台待备份的服务器上需要安装Symantec Backup Exec的Media Server模块、磁带库驱动程序、对应的应用程序备份代理模块。

在备份服务器上，用Symantec Backup Exec提供的配置界面，来制定针对每台服务器的备份策略，策略生效之后，各个服务器便会按照策略中规定的时间、备份源、备份目标来将各自的数据备份到相应的备份目标。这是一个典型的企业数据备份系统案例，数据流经的路径不包括

前端网络，所以属于Frontend Free备份。

16.3.6 数据备份系统案例二

Symantec Netbackup是Symantec 公司的另一个备份产品，与Backup Exec不同的是，Netbackup适合于大型备份系统，支持各种操作系统平台，各个模块可以分别安装在不同操作系统上，由于之间通过TCP/IP协议通信，所以可以屏蔽各种操作系统的不同。而Backup Exec只支持Windows和Netware操作系统（最新的11D版本支持Linux）。NetBackup更加适合异构操作系统平台的备份，因此适合拥有众多不同厂家服务器、不同操作系统的大型企业的备份系统。

提示： 关于NetBackup可参考第16.3.7节的NetBackup配置指南

图16-99所示的是某企业备份系统的拓扑图。

图16-99 某公司IT系统示意图

该企业有四台DB2数据库服务器，主机名分别为：bbpsrv、bbpsr2、srmdb、microcct，皆使用Windows 2000 Advanced Server操作系统。

其上分别安装NetBackup软件的介质服务器模块和磁带库驱动程序。

其中bbpsrv服务器连接有一台SCSI磁带机，microcct服务器连接有两台SCSI磁带机。这四台数据库服务器也是待备份的服务器。

一台备份服务器，主机名为houseprd，使用Windows 2000 Advanced Server操作系统，其上安装有NetBackup软件的Master Server模块（默认

包含了Media Server模块），同时也安装了磁带库驱动程序，并且通过SCSI线缆连接有一台磁带机。

一台HP Ultrium磁带库，包含一个机械手和两个驱动器。

一台IBM 3580磁带库，包含一个机械手和两个驱动器。

在这个备份系统中，由Master Server进行调度，每台DB2数据库服务器都将待备份的数据通过SAN交换机传输给磁带库。由于共有五台服务器使用两台磁带库，所以需要由这五台服务器共享这两台磁带库，为了避免冲突，调度工作统统由Master Server来进行。

16.3.7 NetBackup配置指南

图16-100是在houseped这台Master Server上运行的NetBackup配置工具的主界面。右侧窗口是各种自动配置向导。初次安装完NetBackup软件之后，首先要让NetBackup这个调度员识别到网络上的每台介质服务器，及与其挂接的各种用于备份的存储设备。

图16-100 NetBackup主界面

1. 配置存储设备

初次运行NetBackup配置工具的时候，可以通过右侧窗口的向导“Getting Started”来让Master Server扫描网络上的介质服务器和其上的磁带库设备，并对扫描到的设备以及磁带做一些配置和记录，形成一个初始化环境。

（1）单击Getting Started命令，出现如图16-101所示的对话框。

图16-101 Getting Started

(2) 单击“下一步”按钮，出现如图16-102所示的对话框。

图16-102 Device Configuration

(3) 对话框中提示，初始化过程需要4个步骤，首先扫描所有网络介质服务器可供备份用的设备。单击“下一步”按钮，出现图16-103所示的对话框。

图16-103 扫描介质设备

(4) 单击“下一步”按钮，如图16-104所示，扫描到一个介质服务器。

图16-104 扫描到一个介质服务器

(5) 在图16-104的对话框中，可以选择网络上的所有介质服务器，这样，就可以扫描这些服务器上用于备份的存储设备了，如图16-105所示。

图16-105 扫描到介质服务器上的设备

(6) 扫描结束后，单击“下一步”按钮，出现如图16-106所示的对话框。

图16-106 设备列表

(7) 在对话框中会列出所有检测到的设备的状态，但是图16-106中没有找到任何设备，这可能是由于主机未联入网络或者介质服务器没有安装Media Server模块。单击“下一步”按钮，出现如图16-107所示对话框。

框。该对话框可用来建立一个硬盘目录，这个目录可供备份文件存放，但不是必须的。一般将备份后的数据存入磁带中，用磁盘目录存放备份数据的一个好处就是可以作为一个缓冲，可以设置NetBackup在一定时间后，将这个硬盘上的数据转移到其他备份目标中。

图16-107 建立本地磁盘备份目录

（8）单击“下一步”按钮，出现如图16-108所示的对话框。

图16-108 设置完成

（9）单击“完成”按钮，将进入Volume配置界面，如图16-109所示。

图16-109 配置Volume

（10）所谓Volume，指的就是磁带（在后面会详细介绍）。这一步中，NetBackup会识别所有磁带库中的磁带，并将它们编入默认的Volume Group中供使用。单击“下一步”按钮继续，出现如图16-110所示的对话框。

图16-110 设置Catalog备份

（11）设置Catalog的备份。所谓Catalog，就是NetBackup自身运行所需要的数据，其实NetBackup自身管理维护着一个小型数据库，数据库中保存了NetBackup的所有配置，以及所有磁带、设备、备份策略、过期时间等信息，如果Catalog损坏，则整个NetBackup将会瘫痪，所以备份Catalog自身也是非常重要的。这就像医生给别人治病的同时，自己也要预防疾病一样。NetBackup虽然是一个备份其他数据的软件，但是它也要备份好自身的数据，这一点很好理解。单击“下一步”按钮，如图

16-111所示。

图16-111 Catalog备份向导

(12) 单击“下一步”按钮，如图16-112所示。

图16-112 选择Catalog备份方式

(13) 选择“Online, hot catalog backup”单选按钮，单击“下一步”按钮，如图16-113所示。

图16-113 创建备份策略

(14) 创建一个用于备份Catalog信息的新策略，单击“下一步”按钮，如图16-114所示。

图16-114 输入策略名称

(15) 输入“btest”，单击“下一步”按钮，进入备份方式选择窗口，如图16-115所示。

图16-115 全备和增备方式选择

(16) 选择完全备份或者增量、差量备份。单击“下一步”按钮继续，如图16-116所示。

图16-116 设置Catalog备份日程表

(17) 设置每周进行一次完全备份，每个备份保留期限为两周，两周过后，之前的备份就认为失效，存放备份的磁带可供其他备份使用，如图16-117所示。

图16-117 选择具体Catalog备份日程表

（18）选择具体备份时间，图16-117跨越了所有时间，所以备份可以在任何时间内发生。继续。

（19）设置备份后的Catalog信息存放位置。以及登录操作系统所需的用户认证信息。单击“下一步”按钮继续，如图16-118所示。

图16-118 选择Catalog备份路径及认证信息

（20）设置是否进行邮件通知，如图16-119所示。选择No单选按钮，单击“下一步”按钮继续，如图16-120所示。

图16-119 是否邮件通知 图16-120 完成设置

（21）完成Catalog备份策略向导。单击“下一步”按钮，如图16-121所示。

图16-121 策略列表

选择新创建的策略，单击“下一步”按钮，即可完成环境的初始化操作。

初始化后，Master Server会在Media Server列表中自动加入这些扫描到的介质服务器，并且在Storage Unit中列出扫描到的机械手设备。图16-122所示的是介质服务器列表，图16-123所示的是备份客户端列表。

图16-122 介质服务器列表

图16-123 备份客户端列表

图16-124所示的是存储单元列表。

图16-124 存储单元列表

2. Storage Unit

Storage Unit（存储单元），是一个逻辑上的概念。它表示存储设备中管理一组介质的单元，对于磁带库设备来说，一个机械手就可以掌管属于它的所有磁带，那么一个机械手就是一个存储单元。所以图16-124中，每个磁带库的机械手，都被认为是一个存储单元。可以看到右侧窗口中显示了10个机械手设备，但是物理上只存在两个，这是为何呢？

因为五台服务器共享两个机械手，每台服务器都会识别到两台磁带库的机械手，所以一共是10个机械手设备。实际使用的时候，只允许其中两台服务器同时操纵两台磁带库设备。但是一段时间内，五台服务器均有机会操纵磁带库，这也就是共享磁带库的意义了。

每台磁带库中的可用介质（磁带）也会被自动添加到Media项，如图16-125所示。

图16-125 介质（磁带）列表

图16-125中的Robots，表示物理上存在的磁带库的机械手，所以只有两个。右侧窗口所示的是识别到的所有磁带，每盘磁带都被编了号，以便加以区分。实际上，每盘磁带都会贴有一个条码，机械手扫描这个条码以区分每盘磁带。

3. 卷池（Media Pool）

由于每盘磁带的存储容量有限，如果有备份需要用到多于一盘磁带，则如何分配并在分配后记录这些磁带的使用状况，是个比较麻烦的问题。为了使管理更加方便，NetBackup引入了卷池（Media Pool）的概念。这就像磁盘阵列设备将每个物理磁盘合并，并再分割成更大的Volume或者虚拟磁盘一样，磁带同样可以这样被虚拟化，如图16-126和图16-127所示。

图16-126 卷池（1）

图16-127 卷池（2）

我们看到，Volume Pools项之下的11个卷池，其中名为NetBackup的卷池包含了11盘磁带。而名为bbpserv_filesystem的卷池，只包含了一盘磁带。有了卷池之后，就可以把卷池中的所有磁带，当成一个大的虚拟磁带来看待。我们可以为每个待备份的数据项目分配一个卷池，每次备份的数据只存放在这个卷池中，其他卷池中的磁带不会给这个备份所使用，这样就做到了充分的资源隔离。卷池可以手动创建并且在不冲突的前提下任意添加磁带，如图16-128和图16-129所示。

图16-128 新建卷池

图16-129 输入名称和描述

4. 卷组（Media Group）

这也是一个逻辑上的概念，下图中显示了两个卷组，每个磁带库中的磁带，都放到了一个单独的组中。卷组在实际使用上没有很大的意义。卷组不能手动创建，默认每个机械手就会生成一个卷组，如图16-130所示。

图16-130 卷组

5. Robots（机械手）

如图16-131所示，左侧显示的是机械手。

图16-131 左侧显示的机械手

这个项下面显示了整个备份系统中所存在的物理机械手的数量，右侧窗口中显示了对应机械手所掌管的磁带。

6. Standalone（非共享的机械手）

如图16-132所示是Standalone机械手。

图16-132 Standalone机械手

如果某台介质服务器独立掌管一台磁带库的机械手，而没有共享给其他主机使用，则NetBackup识别到这种设备之后，就会显示在右侧窗口中。本例中没有这种设备。

7. Devices（设备）

这一项列出了整个系统中所有可用于备份的物理存储设备，如图16-133所示。

图16-133 所有可用于备份的介质设备

右侧窗口中的拓扑图显示了两台磁带库和四台独立磁带机，并有连线。带有齿轮标志的为介质服务器，右上方的图标为磁带库，其中还显

示了机械手、磁带槽和两个驱动器，驱动器下面的手托表明这个驱动器为共享驱动器，也就是说其他主机也可以操作这个驱动器。

右侧窗口的下半部分显示了所有逻辑而不是物理设备。由于共享驱动器的原因，本例中的逻辑驱动器变为了12个（三台服务器，每台识别到四个驱动器），再加上独立的磁带机，共有16个驱动器，如图16-134所示。

图16-134 设备列表

同理，逻辑机械手也有6个而不是3个，如图16-135所示。

图16-135 逻辑机械手

如果选中Devices项下面的某台主机设备，便可在右侧窗口的下半部分显示这台主机掌管的驱动器或者机械手，如图16-136所示。

图16-136 每台主机包含的设备

并且在右侧窗口的上半部分，就会突出这台设备所连接的连线，如图16-137所示。

图16-137 拓扑连线

图中黑色连线表示microcct主机目前所连接的设备有：两台独立磁带机驱动器、两台磁带库中的共享驱动器。

8. Standalone Drivers（独立驱动器）

如果某台介质服务器上有自己的独立磁带机（一个驱动器，没有机

械手），则NetBackup识别到之后，就会在这个项目下显示出来。本例中共有4台独立磁带机，如图16-138所示。

图16-138 独立磁带机列表及拓扑

提示：拓扑图中所有独立磁带机的连接线都用黑色加重了。

以上介绍了NetBackup配置工具的一些基本组成。下面通过一个实例来说明如何备份bbpsrv这台服务器上的DB2数据库。

16.3.8 配置DB2数据库备份

1. 建立备份策略

（1）首先建立一个备份策略，命名为“bbpsrv_db2_bak”，如图16-139所示。

图16-139 创建备份策略

（2）单击New Policy命令，输入名称之后，显示如图16-140所示的对话框。

图16-140 新策略窗口

（3）在Policy Type中我们选择“DB2”，使NetBackup调用与DB2备份相关的模块。在Policy Volume Pool下，选中专门为这个备份所创建的卷池“bbpsrv”。

2. 在策略中添加时间表

切换到Schedules选项卡，设置什么时间进行数据备份，以及每次备份最多允许花费多长的时间等。

进入Schedules选项卡，窗口中已经包含了一个名为Default-Application-Backup的Schedule，这个Schedule是备份DB2数据库所必需的，因为备份时需要调用的脚本中的Schedule名称就是Default-Application-Backup。双击这个Schedule，弹出如图16-141所示的对话框。

图16-141 编辑策略属性

其中Type of backup为Application Backup，表明这个Schedule是用于由应用程序自主发起的备份。如果没有这个Schedule，则应用程序就不能调用Netbackup提供的接口而把数据发送给NetBackup，因为策略中没有允许应用程序这么做。

我们所要实现的，不仅仅是手动从应用程序发起备份，而是让NetBackup自动根据设定的时间来备份，所以需要增加一个Schedule。单击下方的New按钮。

将这个Schedule命名为“Auto_Full”，Type of backup选择Automatic Full Backup，如图16-142所示。

图16-142 给日程起名

Frequency选择每天备份一次。Retention（保留）选择将备份保留两周，两周后，对应的磁带就可以被抹掉或者用于其他备份。然后切换到Start Windows选项卡，如图16-143所示。

图16-143 选择具体备份窗口时间

这里设置，每天0点开始备份，凌晨5点结束备份。如果由于某种原因，备份持续了超过5小时，则NetBackup会执行完当前备份。如果还有其他备份需要在这5个小时中执行，则禁止其执行，直到第二天的0点，再接着执行上次未执行的备份，依此类推。

3. 选定需要备份的客户机

接着切换到Clients选项卡，如图16-144所示。

图16-144 设置要备份的客户端

单击New按钮，浏览或者输入要备份的服务器，即bbpsrv这台计算机。期间会提示选择这台计算机的操作系统类型，这里选择Windows 2000。然后切换到Backup Selections选项卡。在前三个选项卡中，已经定义了备份类型、备份发生的时间和持续时间、所需备份的服务器，而唯独缺少了最重要的内容，即备份这台服务器上的哪些东西。在Backup Selections选项卡来完成这个策略的最后一步：定义备份哪些内容。

4. 选择需要备份的内容或者需要执行的脚本

在Clients选项卡中单击New按钮，出现如图16-145所示的界面。

图16-145 选择备份内容或者需要执行的备份脚本

然后单击按钮，来浏览客户机上的文件，如图16-146所示。

图16-146 选择要备份的文件或者要执行的脚本

选中需要备份的目录或者文件之后，单击OK按钮。

本例需要备份的是数据库数据，备份数据库如果只备份数据文件，恢复的时候是不够的，况且，如果是online备份，则必须用数据库自己提供的工具来备份，才会得到可用的镜像，仅仅把数据文件复制一份，这种备份是不能用作恢复的。所以这个例子中，需要在待备份的计算机上运行DB2数据库相关的备份命令来备份数据库。这些命令都存在于一个预先由NetBackup编辑好的批处理脚本文件中。

找到这个文件，其路径位于待备份计算机NetBackup安装目录下：

C: \Program
Files\VERITAS\NetBackup\DbExt\DB2\db2_backup_db_online.cmd

选中这个文件，单击OK按钮，结果如图16-147所示。

图16-147 选择要执行的脚本

可以看到，NetBackup已经识别出这个脚本，左侧的图标已经变为。

脚本的内容如下：

```
@REM $Revision: 1.2 $  
@REM bcpyright  
@REM *****  
@REM * $VRTScprght: Copyright 1993 - 2003 VERITAS Software Co  
@REM *****  
@REM ecpyright  
@REM -----  
@REM
```

```
@REM This script is provided as an example. See the instructions
@REM for making customizations to work within your environment
@REM
@REM Please copy this script to a safe location before customizing
@REM Modifications to the original files will be lost during updates
@REM
@REM This script performs an online backup of the database. An online backup
@REM requires that the database is configured for forward recovery (see
@REM DB2 USEREXIT and LOGRETAIN settings). DB2 users can remain connected
@REM while performing an online backup.
@REM
@REM To back up a database or a database partition, the user must have
@REM SYSCTRL, or SYSMANT authority.
@REM -----
@echo off
@setlocal
@REM !!!!! START CUSTOMIZATIONS !!!!!
@REM
@REM The following changes need to be made to make this script work in your
@REM environment:
@REM
@REM -----
@REM (1) NetBackup for DB2 shared library:
@REM -----
@REM     This is the NetBackup library that backs up and restores the
@REM     Set db2_nbllib below to the correct NetBackup library
@REM
```



```

@REM Example: @set db2_nblib=C:\progra~1\veritas\netbackup\bi
@set db2_nblib=
@echo db2_nblib = %db2_nblib%
@REM -----
@REM (2) DB2 home directory (the system catalog node):
@REM -----
@REM     This is the DB2 home directory where DB2 is installed
@REM     Set db2_home to DB2 home directory
@REM
@REM Example: @set db2_home=D:\sqllib
@set db2_home=
@echo db2_home = %db2_home%
@REM -----
@REM (3) Database to backup:
@REM -----
@REM     Set db2_name to the name of the database to backup:
@REM
@REM Example: @set db2_name=SAMPLE
@set db2_name=
@echo db2_name = %db2_name%
@REM -----
@REM (4) Multiple Sessions:
@REM -----
@REM     Concurrency can improve backup performance of large d
@REM     Multiple sessions are used to perform the backup, wit
@REM     backing up a subset of the database. The sessions open
@REM     concurrently, reducing the overall time to backup the

```

```

@REM      This approach assumes there are adequate resources av
@REM      multiple tape devices and/or multiplexing enabled.
@REM
@REM      For more information on configuring NetBackup multipl
@REM      refer to the "Veritas NetBackup System Administrator'
@REM
@REM      If using multiple sessions change db2_sessions to use
@REM
@REM Example: @set db2_sessions="OPEN 2 SESSIONS WITH 4 BUFFE
@set db2_sessions=
@REM !!!!! END CUSTOMIZATIONS !!!!!
@REM -----
@REM Exit now if the sample script has not been customized
@REM -----
if "%db2_name%" == "" goto custom_err_msg
@REM -----
@REM These environmental variables are created by Netbackup (
@REM -----
@echo DB2_POLICY = %DB2_POLICY%
@echo DB2_SCHED = %DB2_SCHED%
@echo DB2_CLIENT = %DB2_CLIENT%
@echo DB2_SERVER = %DB2_SERVER%
@echo DB2_USER_INITIATED = %DB2_USER_INITIATED%
@echo DB2_FULL = %DB2_FULL%
@echo DB2_CINC = %DB2_CINC%
@echo DB2_INCR = %DB2_INCR%
@echo DB2_SCHEDULED = %DB2_SCHEDULED%

```

```

@echo STATUS_FILE = %STATUS_FILE%

@REM -----
@REM Type of Backup:
@REM -----
@REM     NetBackup policies for DB2 recognize different
@REM     backup types, i.e. full, cumulative, and differential
@REM     For more information on NetBackup backup types, pleas
@REM     NetBackup for DB2 System Administrator's Guide.
@REM
@REM     Use NetBackup variables to set DB2 full or incrementa
@REM
@set db2_action=
if "%DB2_FULL%" == "1" @set db2_action=ONLINE
if "%DB2_CINC%" == "1" @set db2_action=ONLINE INCREMENTAL
if "%DB2_INCR%" == "1" @set db2_action=ONLINE INCREMENTAL DEL
@echo db2_action = %db2_action%

@REM -----
@REM Actual command that will be used to execute a backup
@REM Note: the parameters /c /w /i and db2 should be used wit
@REM Without them, NetBackup job monitor may not function pro
@REM -----
@set CMD_FILE=%temp%\cmd_file
@echo CMD_FILE = %CMD_FILE%
@set CMD_LINE=%db2_home%\bin\db2cmd.exe /c /w /i db2 -f %CMD_
@echo CMD_LINE = %CMD_LINE%
@echo BACKUP DATABASE %db2_name% %db2_action% LOAD %db2_nblib
@echo BACKUP DATABASE %db2_name% %db2_action% LOAD %db2_nblib

```

```

@REM -----
@REM Execute the command
@REM -----
@echo Executing CMD=%CMD_LINE%
%CMD_LINE%
@REM Successful Backup
if errorlevel 1 goto errormsg
echo BACKUP SUCCESSFUL
if "%STATUS_FILE%" == "" goto end
if exist "%STATUS_FILE%" echo 0 > "%STATUS_FILE%"
goto end
:custom_err_msg
echo This script must be customized for proper operation in y
@REM Backup command unsuccessful
:errormsg
echo Execution of BACKUP command FAILED - exiting
if "%STATUS_FILE%" == "" goto end
if exist "%STATUS_FILE%" echo 1 > "%STATUS_FILE%"
:end
@endlocal

```

经过这样的配置之后，在每天的0点，Master Server便会发送指令给bbpsrv上的NetBackup客户端，让它执行这个脚本。此脚本中的命令会告诉DB2数据库备份数据库并且调用一个DLL链接库文件，将数据通过SAN网络发送到相应卷池所在的磁带库上，从而被写入磁带。

5. 监控备份执行状况

可以通过下面的方法监控备份执行的状况，如图16-148所示。

图16-148 监控备份状态

右侧窗口的下半部分显示了备份运行的状况，如果成功备份，则显示一个蓝色小人成功举起的图标；如果备份正在运行，则显示一个绿色小人正在奔跑的图标；如果备份任务正在等待，则显示三个绿色小人排队图标；如果任务失败，则显示圆形红色差号。

对于其他服务器，可以用同样的步骤来备份，这里就不再详述了。

16.4 与业务应用相结合的快照备份和容灾

16.3一节末尾所介绍的备份DB2数据库的方法，其实只是调用了DB2自身所提供的API而已，DB2将数据通过API传输给备份软件代理，然后备份软件将收到的数据写入对应的存储介质中。包括各大备份软件对Oracle的备份，其实也都是这样，将Oracle置于热备模式下，然后将其对应的数据文件以及配套日志复制出来保存，或者利用RMAN提供的API来实现数据备份。这些备份方式都需要影响到业务主机，都需要由业务主机将对应的数据提取出来传送给备份代理，然后备份代理程序将其写入存储介质。这种备份所耗费的时间会很长，因为涉及到实际的数据移动，所以备份过程中对主机资源会有持续的耗费。

而如果能够直接在这些应用程序存储数据的磁盘阵列设备中对相应的数据做备份的话，那么就完全可以实现Server-Free了。在磁盘阵列设备中如何对数据做备份呢？磁盘阵列所能做的就是直接将整个LUN中的数据传输到另外的位置。由于LUN中的数据是不断变化的，所以阵列必须先对这个LUN做一份快照，然后将这份快照数据通过某种方式传输到备份目标存储空间中。在做快照时，必须保证应用程序层面的一致性，这也就意味着应用程序需要将其缓存中的脏数据或者日志刷盘之后，阵列再对相应的LUN做快照，此时做出来的快照才是应用层一致的。所以，主机端需要有一个主机代理程序来负责通知对应的应用程序，调用应用程序提供的API让其处于归档模式或者静默模式，之后代理通知阵列做快照；或者可以利用VSS框架来实现对多种应用的一致性支持。快照做完之后，这份快照就相当于一份一致的备份数据集了，此时可以将其通过dump的格式直接写入磁带中，或者通过网络将其传输到另一台阵列的LUN中保存。对于后续的备份，可以实现增量备份，比如再次做

快照，将这份最新快照与上一次备份时产生的快照做比对，比对快照中的bitmap，将两份bitmap做OR运算，得出的结果bitmap中为1的则标示变化的块，后续只要将这些变化的块同步到远程阵列或者利用dump格式再次写入磁带即可。

上述过程只体现了“与业务应用相结合”这个思想的一部分，也就是保证数据一致性。这个思想的另一部分则是体现在对业务底层所使用的存储空间的全面管理，以及对业务的细粒度备份和恢复方面。

如图16-149所示为一个上文中提到的主机端代理（Driver Agent，作者自创的名词，并非特指某款产品名称）的架构图，这个代理其实就是快照代理的一个升级版。可以从图中看到它所支持的功能，作为一个高级卷管理角色，不但可以直接从阵列端要存储空间，而不是在阵列端配置然后映射给主机，还可以创建快照、挂载这些快照等。如图16-150所示为App Agent（作者自创的名词，并非特指某款产品名称）的架构图，其可以与各种应用程序以及VSS框架交互，可以感知应用程序的内部信息，比如实例名、数据文件和所在的存储空间等，并且可以将对应的应用数据迁移到其他存储位置。这个代理通过调用Driver Agent的功能来实现对应用系统数据的恢复，可以做到细粒度的备份和恢复，比如只备份某个数据库实例，只备份某个Exchange Mailbox，只恢复某个Mailbox等。

图16-149 Driver Agent软件架构图

图16-150 Driver Agent与APP Agent交互架构图

目前这种备份架构对应的产品有比如EMC的Replication Manager以及NetApp的SnapManager，以及华为赛门铁克的HostAgent产品。

另外，在对虚拟机的备份过程中，一些备份软件可以利用Vmware所提供的一些存储API实现一些高级功能，比如某备份软件厂商所实现的Active Block Mapping功能。我们知道，有些备份软件是直接将Vmware宿主机上VMFS下面的VMDK虚拟机虚拟磁盘文件备份下来的，而这些磁盘如果是被NTFS文件系统所格式化并且管理使用的，那么NTFS中实际占用虚拟磁盘空间的文件可能并没有占用全部虚拟磁盘的空间，那么此时备份软件是无法感知到这一点的，它只能将整个VMDK文件备份下来，浪费了大量存储空间和网络带宽。

而这个Active Block Mapping功能通过调用Vmware所提供的API，检测VMDK中实际被NTFS文件系统占用的块，然后将NTFS中的剩余空间填0，此处“填0”并不是真的去向其中写入0x00，而是使用元数据记录这些0x00的位置以及长度。我们知道，所有文件系统都会使用bitmap来记录当前文件系统所在的底层磁盘/卷空间块的占用和空闲状态，备份软件只要将这份位图从对应的虚拟机文件中提取出来便可知道到底这个文件中哪些区段是空闲的，然后备份软件在备份这个VMDK文件时只需要将文件中被NTFS所占用的部分从底层磁盘中读取出来连续存放即可。当恢复的时候，系统会根据这份位图来实时地向对应的VMDK恢复流中填入0（在内存中填入），之后将数据发送给恢复目标介质。这样就大大降低了磁盘IO压力以及网络传输的带宽压力。

第17章 愚公移山——大话数据容灾

- 本地站点
- 远程站点
- 数据通路
- 同步复制
- 异步复制
- 基于主机的数据复制
- 基于存储的数据复制

数据备份系统只能保证数据被安全地复制了一份，但是一旦生产系统发生故障，比如服务器磁盘损坏致使数据无法读写、主板损坏造成直接无法开机或者机房火灾等意外事件，我们必须将备份的数据尽快地恢复到生产系统中继续生产，这个动作就叫做容灾。容灾可以分为四个级别：数据级容灾，也就是只考虑将生产站点的数据如何同步到远程站点即可；与应用结合的数据级容灾，也就是可以保证对应应用程序数据一致性的数据同步，以及可感知应用层数据结构的、有选择的同步部分关键重要数据的数据容灾；应用级容灾，也就是灾难发生时，不仅可以保证原本生产站点的数据在灾备份站点可用，而且还要保证原生产系统中的应用系统比如数据库、邮件服务在灾备份站点也可用；业务级容灾，

除了保证数据、应用系统在灾备份站点可用之外，还要保障整个企业的业务系统仍对外可用，这里面就包含了IT系统可用、IT管理部门可用、业务逻辑部门可用、对外服务部门可用等，是最终层次的容灾。

17.1 容灾概述

有些事件中，很多公司就是因为没有远程容灾系统，导致数据全部毁于一旦，客户数据丢失、公司倒闭，受损失的不仅是公司，还有客户。如果要充分保障系统和数据的安全，只是在本地将数据进行备份还远远不够，还必须在远程地点建立另外一个系统，并包含当前生产系统的全部数据备份。这样在本地系统发生故障的时候，远程备份系统可以启动，继续生产。

要实现这样一个系统，首先，应保证主生产系统的所有数据实时地传输到远程备份系统。其次，主系统发生故障之后，必须将应用程序也切换到远程备份系统上继续运行。应用程序是一个企业生产流程的代码化表示，只有应用程序正在运行，这个企业才处于生产过程中，而应用程序的成功运行，又必须依赖于底层数据。

俗话说，巧妇难为无米之炊。我们的应用程序，比如Exchange邮件转发系统、SAP企业ERP系统、Lotus Notes办公自动化系统等，这些就好比巧妇，而保存在磁盘上的数据，比如用户的邮件、ERP系统的数据库文件、办公自动化系统自身的数据文件等，就好比大米，巧妇用她高超的厨艺，将大米做成熟饭，供消费者购买。

这就是一个企业生产的基本雏形，企业（巧妇）用应用程序（高超的厨艺）来处理各种数据（大米），最终生成新的数据（米饭），供消费者购买。而巧妇所利用的锅碗瓢盆、水、电、煤气等也必不可少，比如服务器、硬盘、网络通信设施、电源等这些IT系统必要组件。下面来对比一下厨房和IT系统机房，如表17-1所示。

表17-1 厨房与机房

生产者厨师，用生产工具来加工生产资料，获得产品。同样，各种应用程序，运行在服务器上，将各种原始数据加工修改，产生客户需要的信息。二者在本质上是相同的。

基于这个生产模型，我们把一个企业IT生产系统划分为4个组件：生产资料、生产工具、生产者、产品。要实现整个IT系统的容灾，那么必然要实现上述所有4个组件的容灾。然而，IT系统的产品和原始数据往往都存放在同一位置，比如同一个卷、同一台盘阵等。本章不描述产品的容灾，因为其与生产资料的容灾本质是相同的。

思考： 厨房的容灾。大家不要笑，厨房容灾，这个名词是不是太荒唐了。现在貌似是的，但是战争年代，厨房容灾将是必备的。人是铁饭是钢，关键时刻看厨房。我想厨房容灾这个话题，广大读者应该比IT工作者的办法更多了。比如，在另外一个隐蔽地点建立一个厨房，柴米油盐酱醋茶、锅碗瓢盆炉灶勺都储存到这个厨房中作为储备粮和储备工具，一旦当前厨房被敌人摧毁，立即启用备用厨房，厨师全部转移到备用厨房继续做饭。这没啥难度。的确，谁都可以想出这样的办法。那么我们看看能否用这个思想来建立IT系统的容灾。

生产工具的容灾

像厨房容灾一样，在另一个地点建立一个IT机房，服务器、网络设施、磁盘阵列设施等一应俱全。当然，出于成本因素，备用地点的设备不一定非要与主系统中的生产工具规格和性能完全相同，在性能和容量上的要求可以适当降低，但至少能满足生产需求。

下面将主要讲解生产资料的容灾和生产者的容灾。

17.2 生产资料容灾——原始数据的容灾

IT系统的生产资料，即各种原始录入数据。它和实物化的生产资料比如大米，有很大不同。

第一，IT数据是可以任意复制，并可以复制多份的数据。

第二，IT系统数据是不断变化的，在生产的同时，原始数据将会不断地变化，甚至产品数据会覆盖原始数据。

基于IT数据的这两个特点，在IT系统的生产资料容灾方面，需要注意以下两点。

第一，不可能像储存大米一样，把某时刻的原始数据复制到备用系统中就不管了，因为这份数据是不断在变化中的，我们需要把变化实时地同步到备用系统中，只要主系统数据变化了，备用系统的数据也要跟着变化。

第二，数据必须至少保留额外的一份。因为大米没有了，可以再购买，每次购买的大米也可以一样的。但是如果数据没有了，就不可再生，这个企业就要面临倒闭。所以在实现容灾的同时，还必须做好数据备份工作，将数据备份到磁带或者其他备份目标中保存。基于IT系统数据是不断变化的，所以需要尽量保存这份数据的最后状态，比如，一天备份一次。如果数据量很小，甚至可以一天备份多次，这样可以充分保证备份的数据与当前的数据相差最少。

有了以上两点的保证，就可以像厨房容灾一样，来设计IT系统的生产资料容灾了。以下是一个设计好的例子。

(1) 用网络来连接本地系统和备用系统，先将本地系统某时刻的数据，实时传送到备用系统。

(2) 传输结束后，再将从这个时刻之后的所有变化的数据，同步到备用系统。

(3) 此后，只要本地系统有数据变化，则立即将变化的数据传送到备用系统，使备用系统数据发生相同的变化。

在这个基础上，还需要在本地系统中对数据做额外备份，即备份到离线（如磁带等）介质上，做到时刻保留一份额外的数据。有条件的，还可以将备份好的磁带运送到备份站点去，这样就充分保证了主站点一旦发生火灾等全损型故障后的数据冗余度。

生产资料的容灾，是容灾系统中最重要的一個组件，因为有了生产资料，即使有生产工具、生产者，也无法进行生产；而如果没有了生产工具和生产者，比如服务器、应用软件等，则可以很容易购买到。

将生产数据通过网络实时传送到备用系统，要实现这个目的，要怎么来设计呢？我们来看个拓扑图，如图17-1所示。

图17-1 两种数据通路的位置

左边的主站点和右边的备份站点存在相同的生产工具。既然要使主站点和备用站点之间必须通过网络来连接，所以只要知道究竟在哪个网络设施上进行连接。系统中有两个网络通信设施，一个是前端以太网，另一个是后端SAN网络（也有可能是以太网，即服务器通过iSCSI协议连接到后端磁盘阵列），我们究竟是连接二者的前端网络还是连接后端网络呢（图中标注的两条路径）？答案是，两者都可以。

17.2.1 通过主机软件实现前端专用网络或者前端公用网络同步

这种方式利用的就是图17-1中的标注1所示的路径。

(1) 主站点和备份站点的前端以太网络，均通过路由器连接至电信部门的专线或者Internet网络上。

(2) 主站点上变化的数据，经过前端以太网交换机，然后通过路由器，传送给电信部门的网络交换路由机组中，经过层层交换或路由，传输到备份站点的路由器。

(3) 然后经过交换机传送到备份站点的相应服务器上。

(4) 服务器收到数据后，写入后端的磁盘阵列上。

提示： 如果为了连接而接入Internet网络，则最好做成VPN模式，在隧道的基础上，如果追求数据安全性，则需要配置加密模式的VPN。如果对数据同步的实时性要求不高，而数据量又很大的情况下，主站点和备份站点都接入100Mb/s的Internet网络，反而会得到很大的实惠，特别是备份站点和主站点在相同城市的情况下，这样主站点数据路由到电信部门的设备之后，经过很少的设备就会到达备份站点，而且能保证很大的实际带宽。如果是利用窄带专线接入专网，虽然可以保证这条链路带宽独享，但是毕竟带宽低。所以专线可以保证数据同步的实时性，但是不适合大数据量的传输。

因为这种方式同步数据，需要经过前端网络，所以实现这个功能，还需要在距离前端网络最近的主机设备上实现，也就是服务器群上来实

现。

思考： 为何不能直接在网络交换设备上或者路由器上来实现呢？

第一，网络设备一般是没有灵活的程序载入运行能力的，网络设备上运行的程序都是预先固化到芯片中的程序，一般不可修改，更不用说再加入另外的程序来执行了。

第二，数据存在于服务器，或者连接服务器的磁盘阵列上，网络设备若想从服务器上提取数据，则必须通过调用操作系统提供的相关程序接口。而跨网络传输数据的现成接口，只有网络文件系统，所以还需要将所有数据卷都共享成为NAS模式，但这样过于繁冗，所以我们必须要在每台有数据备份的服务器上安装一种软件来实现数据的同步，将这种软件安装在生产者，也就是应用服务器上，然后在服务器上提取生产资料和产品。

不管生产资料和产品存在于服务器本地磁盘，还是存在于后端的磁盘阵列上，对于服务器操作系统来说，都是一个个的目录。在第15章的最后曾经描述过操作系统的目录虚拟化，不管底层用的是什么设备来存储数据，也不管数据存放在网络上还是本地磁盘上，最终操作系统都将这些位置虚拟成目录，比如Windows的C盘、D盘，或者UNIX下的/mnt、/mountpoint等。

通过这种软件可以直接监视这些目录中数据的变化，只要有变化的数据，就提取出来通过网络传送到远端服务器上，远端服务器同样需要安装这种软件的接收端模块来接收数据，并写入远端备份站点上相应服务器的相同目录。

这种方式利用了前端网络进行数据同步。一般前端网络相对后端网络速度来说是比较低的网络，而且是客户用来访问服务器的必经之路，所以它的资源是比较宝贵的。另外，前端网络一般都是以太网，相对廉价，而且容易整合到企业大网络中，从而接入电信部门的网络，所以适合基于TCP/IP协议的远距离传输，比如大于100公里的范围，甚至跨国界的Internet范围内传输。

下面来看一下这种方式下的数据流经路径：

本地磁盘阵列（或者本机磁盘）→本地后端网络交换设施→本地服务器内存→本地前端网络→电信交换机组→远端前端网络→远端服务器内存→远端后端网络交换设施→远端磁盘阵列（或者远端本机磁盘）。

如果数据源在本机磁盘，则会跳过本地后端网络交换设施，直接到服务器内存。如果数据是直接内存中生成的，则需要写入本地一份，同时发送给远端一份保存。其中，在“本地磁盘阵列→本地后端网络交换设施→本地服务器内存”这段路径上，数据是通过FCP协议（SCSI over FC协议）进行打包传送的；在“本地前端网络→电信交换机组→远端前端网络”这段路径上，数据是通过TCP/IP协议传送的。FCP协议运行在后端高速网络的保障之上，而TCP/IP协议运行在使用前端低速网络的设备上，保障数据传输，二者各得其所，充分发挥着各自的作用。

图17-2所示的路径即是数据流动的路径。服务器上的涡轮泵表示数据同步软件，它从本地提取数据，并将数据源源不断地发送给远端，远端服务器上的涡轮泵将收到的数据源源不断地写入存储设备。

图17-2 经前端网络备份数据

这种方式的数据同步，一般都是文件级同步，即同步软件只检测文

件这一层的数据变化，或者每当主服务器针对某个文件做写入时，写入数据会同时发送给备用服务器。而对底层卷的数据块的变化，不做同步，除非数据块的变化造成了对应的文件的逻辑数据变化。

Veritas Volume Replicator软件介绍

Veritas Volume Replicator（VVR），是Veritas公司容灾套件Storage Foundation系列软件中的一个模块，它的作用非常专一，就是将本地某个卷上的数据变化，通过前端IP网络复制到远端对应的卷上，而且保证数据变化发生的顺序不被打乱，完全按照本地的IO发生顺序在异地按照相同的顺序重现这个IO。

VVR支持耗费带宽调整功能，控制对网络带宽的使用，在业务繁忙时可以降低发送速度来减少对网络带宽的耗费，并且可以针对不同的同步流设定各自的带宽。支持异步和同步复制（在后面会介绍）。在网络发生故障的时候，可以自动将复制模式从同步切换到异步，以减少对主机业务的影响。一旦复制断开，VVR可以记录主站点自从断开之后的数据变化，待连接重新建立之后，立即复制这些变化的数据，而不需要对两边数据进行重新比对或者全部重新复制。

提示： 类似的软件还有很多，比如Double Take、Legato，国产同步软件InfoCore Replicator等，它们的构思都是一样的，只不过实现方式和效果上有所不同。

后文将介绍两个使用这种方式来同步数据的案例。

17.2.2 案例：DB2数据的HADR组件容灾

这个案例是笔者实施的一个DB2数据库容灾案例，它利用了运行在

主机和备份机上的一个数据库软件模块（即HADR），来实现两端的数据同步，主机和备份机之间使用基于以太网的TCP/IP协议连接。这个案例同步的数据不是卷上的原始数据，而是一种对数据操作的描述，即数据库日志，比如：“在D盘创建一个表空间数据文件，名称testspace，大小500MB”。

这就是一条日志，主机只需要把这句话告诉备份机即可，而不需要传输500MB的数据。备份机收到日志后，便会在备份机的磁盘上重做（replay）这些操作，达到与直接同步卷上数据殊途同归的效果。

1. HADR

其全称为High Availability Disaster Recovery，它是DB2数据库级别的高可用性数据复制机制，最初被应用于Informix数据库系统中，称为High Availability Data Replication（HDR），是Share-Nothing方式容灾的典型代表。

提示： 在IBM收购Informix之后，这项技术就应用到了新的DB2发行版中。

一个HADR环境需要两台数据库服务器：主数据库服务器（primary）和备用数据库服务器（standby）。

- 当主数据库中发生事务操作时，会同时将日志文件通过TCP/IP协议传送到备用数据库服务器，然后备用数据库对接收到的日志文件进行重放（Replay），从而保持与主数据库的一致性。
- 当主数据库发生故障时，备用数据库服务器可以接管主数据库服务器的事务处理。此时，备用数据库服务器作为新的主数据库服

务器进行数据库的读写操作，而客户端应用程序的数据库连接可以通过自动客户端重新路由（Automatic Client Reroute）机制转移到新的主服务器。

- 当原来的主数据库服务器被修复后，又可以作为新的备用数据库服务器加入HADR。通过这种机制，DB2 UDB实现了数据库的故障恢复和高可用性，最大限度地避免了数据丢失。图17-3为DB2 HADR的工作原理图。

图17-3 DB2 HADR工作原理图

注意：处于备用角色的数据库不能被访问。

HADR有三种同步方式。

1) SYNC（同步）

此方式可以尽可能地避免事务丢失，但在三种方式中，使用此方式会导致事务响应时间最长。在此方式中，仅当日志已写入主数据库上的日志文件，而且主数据库已接收到来自备用数据库的应答，确定日志也已写入备用数据库上的日志文件时，方才认为日志写入是成功的。保证日志数据同时存储在这两处。

如果备用数据库在重放日志记录之前崩溃，则它下次启动时，可从其本地日志文件中检索和重放这些记录。如果主数据库发生故障，故障转移至备用数据库，可以保证任何已在主数据库上落实的事务，也在备用数据库上落实。故障转移操作之后，当客户机重新与新的主数据库连接时，可能会有在新主数据库上已落实的事务，对于原始主数据库却从未报告为已落实。当主数据库在处理来自备用数据库的应答消息之前出现故障时，即会出现此种情况。客户机应用程序应考虑查询数据库以确

定是否存在此类事务。

如果主数据库失去与备用数据库的连接，则不再认为这些数据库处于对等状态，而且将不阻止事务等待来自备用数据库的应答。如果在数据库断开连接时执行故障转移操作，则不保证所有已在主数据库上落实的事务将出现在备用数据库上。

当数据库处于对等状态时，如果主数据库发生故障，则可以在故障转移操作之后，作为备用数据库重新加入HADR对。因为在主数据库接收到来自备用数据库的应答，确认日志已写入备用数据库上的日志文件之前，不认为事务已落实，所以主数据库上的日志顺序将与备用数据库上的日志顺序相同。原始主数据库（现在是备用数据库）只需要通过重放自从故障转移操作以来，在新的主数据库上生成的新日志记录来进行同步更新。

如果主数据库发生故障时并未处于对等状态，则其日志顺序可能与备用数据库上的日志顺序不同。如果必须执行故障转移操作，主数据库和备用数据库上的日志顺序可能不同，因为在故障转移之后，备用数据库启动自己的日志顺序。因为无法撤销某些操作（比如，删除表），所以不可能将主数据库回复到创建新的日志顺序的时间点。

如果日志顺序不同，当在原始主数据库上发出指定了AS STANDBY选项的START HADR命令时，将返回错误消息。如果原始主数据库成功地重新加入HADR对，则可以通过发出未指定BY FORCE选项的TAKEOVER HADR命令来完成数据库的故障恢复。如果原始主数据库无法重新加入HADR对，则可以通过复原新的主数据库的备份映像来将此数据库重新初始化为备用数据库。

2) NEARSYNC (接近同步)

此方式具有比同步方式更短的事务响应时间，但针对事务丢失提供的保护也很少。在此方式中，仅当日志记录已写入主数据库上的日志文件，而且主数据库已接收到来自备用系统的应答，确定日志也已写入备用系统上的主存储器时，才认为日志写入是成功的。仅当两处同时发生故障，并且目标位置未将接收到的所有日志数据转移至非易失性存储器时，才会出现数据的丢失。

如果备用数据库在将日志记录从存储器复制到磁盘之前崩溃，则备用数据库上将丢失日志记录。通常，当备用数据库重新启动时，它可以从主数据库中获取丢失的日志记录。然而，如果主数据库或网络上的故障使检索无法进行，并且需要故障转移时，日志记录将不会出现在备用数据库上，而且与这些日志记录相关联的事务将不会出现在备用数据库上。

如果事务丢失，则在故障转移操作之后，新的主数据库与原始主数据库不相同。客户机应用程序应该考虑重新提交这些事务，以便使应用程序状态保持最新。

当主数据库和备用数据库处于对等状态时，如果主数据库发生故障，则在没有使用完全复原操作重新初始化的情况下，原始主数据库可能无法作为备用数据库重新加入HADR对。

如果故障转移涉及丢失的日志记录（因为主数据库和备用数据库已发生故障），主数据库和备用数据库上的日志顺序将会不同，并且在未执行复原操作的情况下，重新启动原始主数据库以作为备用数据库的尝试将会失败。如果原始主数据库成功地重新加入HADR对，则可以通过

发出未指定BY FORCE选项的TAKEOVER HADR命令来完成数据库的故障恢复。

如果原始主数据库无法重新加入HADR对，则可以通过复原新的主数据库的备份映像来将其重新初始化为备用数据库。

提示： 局域网环境一般采用NEARSYNC方式进行同步。

3) ASYNC（异步）

提示： 如果主系统发生故障，此方式发生事务丢失的几率最高。在三种方式之中，此方式的事务响应时间也是最短的。

在此方式中，只有当日志记录已写入主数据库上的日志文件，而且已将此记录传递给主系统主机的TCP层时，才认为日志写入是成功的。因为主系统不会等待来自备用系统的应答，所以当事务仍处于正在传入备用系统的过程中时，可能会认为事务已落实。

主数据库主机上、网络上或备用数据库上的故障可能导致传送中的日志文件丢失。如果主数据库可用，则会在此对重新建立连接时，将丢失的日志文件重新发送至备用数据库。然而，如果在丢失日志文件时要求执行故障转移操作，则日志文件和相关联的事务都将不会到达备用数据库。丢失的日志记录和主数据库上的故障会导致事务的永久丢失。

如果事务丢失，则在故障转移操作之后，新的主数据库与原始主数据库不是完全相同的。客户机应用程序应该考虑重新提交这些事务，以便使应用程序状态保持最新。

当主数据库和备用数据库处于对等状态时，如果主数据库发生故障，则在没有使用完全复原操作重新初始化的情况下，原始主数据库可

能无法作为备用数据库重新加入HADR对。

如果故障转移涉及丢失的日志记录，主数据库和备用数据库上的日志顺序将会不同，并且重新启动原始主数据库以作为备用数据库的尝试将失败。因为，如果在异步方式中发生故障转移，日志记录更有可能丢失，所以主数据库不能重新加入HADR对的可能性也更大。如果原始主数据库成功地重新加入HADR对，则可以通过发出未指定BY FORCE选项的TAKEOVERHADR命令来完成数据库的故障恢复；如果原始主数据库无法重新加入HADR对，则可以通过复原新的主数据库的备份映像将此数据库重新初始化为备用数据库。

2. BBP系统结构

某企业目前有一套物流BBP系统，基于SAP构建，SAP后台数据库使用的是DB2 v8.2。现有一台闲置服务器（IP：192.168.100.23），使用这台服务器作为HADR系统的备份节点，已经上线运行的BBP服务器（IP：192.168.100.231）作为系统的主节点。B2B系统目前的拓扑与实现HADR之后的拓扑图如图17-4所示。

图17-4 B2B系统目前的拓扑与实现HADR之后的拓扑图

HADR系统将目前闲置的备份机充分利用了起来，一旦主节点发生故障，可以立即手动切换到备份节点，与此同时，客户端会自动重新连接到备份机，使得生产继续进行。故障的主机可以离线进行故障恢复。恢复之后可以加入HADR组，并且重新接管所有应用。表17-2简要说明了实施HADR的过程。

表17-2 在上述环境下实施HADR的简单过程

■ Hadr_pri.bat的内容:

```
db2 "UPDATE DB CFG FOR bbp USING HADR_LOCAL_HOST 192.168.100.  
db2 "UPDATE DB CFG FOR bbp USING HADR_LOCAL_SVC 64000"  
db2 "UPDATE DB CFG FOR bbp USING HADR_REMOTE_HOST 192.168.100  
db2 "UPDATE DB CFG FOR bbp USING HADR_REMOTE_SVC 64001"  
db2 "UPDATE DB CFG FOR bbp USING HADR_REMOTE_INST db2bbp"  
db2 "UPDATE DB CFG FOR bbp USING HADR_SYNCMODE NEARSYNC"  
db2 "UPDATE DB CFG FOR bbp USING HADR_TIMEOUT 60"  
db2 "update alternate server for db bbp using hostname 192.16  
db2 "update db cfg for bbp using logindexbuild on"  
db2 "update db cfg for bbp using indexrec restart"
```

■ hadr_std.bat的内容:

```
db2 "UPDATE DB CFG FOR bbp USING HADR_LOCAL_HOST 192.168.100.  
db2 "UPDATE DB CFG FOR bbp USING HADR_LOCAL_SVC 64001"  
db2 "UPDATE DB CFG FOR bbp USING HADR_REMOTE_HOST 192.168.100  
db2 "UPDATE DB CFG FOR bbp USING HADR_REMOTE_SVC 64000"  
db2 "UPDATE DB CFG FOR bbp USING HADR_REMOTE_INST db2bbp"  
db2 "UPDATE DB CFG FOR bbp USING HADR_SYNCMODE NEARSYNC"  
db2 "UPDATE DB CFG FOR bbp USING HADR_TIMEOUT 60"  
db2 "update alternate server for db bbp using hostname 192.16  
db2 "update db cfg for bbp using logindexbuild on"  
db2 "update db cfg for bbp using indexrec restart"
```

完成配置之后，在主机或者备份机上的DB2命令行环境中输入以下命令：

```
Db2 get snapshot for db on bbb
```

即可以查看HADR的运行状态，如图17-5所示。

图17-5 HADR的运行状态

17.2.3 通过主机软件实现后端专用网络同步

用这种方式来同步数据，数据不会流经前端网络，而全部通过后端网络传输到备份站点对应的存储设备中。这就需要将主站点的后端网络设施和备份站点的后端网络设备连接起来。或者直接通过裸光纤连接两台SAN交换机；又或者租用电信部门的光缆专线。

如果用前者连接两个站点，那要求两个站点之间的道路上可以自己布线，比如一个大院内，可以自主布线，不需要经过市政部门干预，这样便可以直接连接两端的SAN交换机，直接承载FC协议了。否则，如果两个站点跨越了很远的距离，那么就必须使用后者，也就是租用电信部门的光缆，但是这条光缆上的数据必须符合电信部门传输设备所使用的协议。后者需要添加额外的协议转换设备，两个站点各一个，如图17-6所示。

图17-6 连接两个站点的后端网络

现在电信部门的光纤专线一般为SDH传输方式，接入到用户端的时候，一般将信号调制成E1、OC3等编码方式，所以必须将FC协议承载于这些协议之上，也就是我们在前面第13章中所描述的Protocol Over Protocol模型，完成这个动作的，就是协议转换器。

协议转换器在一端按照某种协议的逻辑进行工作，而在另一端则按

照另一种协议的逻辑进行工作，把数据从一端接收过来，经过协议转换，以另一种协议的逻辑发送出去，到达对端后，再进行相反的动作。

有些路由器则直接在其内部集成了各种协议转换器，可以说路由器就是一种协议转换器。我们可以看一下机房中的网络路由器，上面有各种各样的接口，为何不清一色都是RJ-45以太网接口呢？

因为路由器不只是路由以太网数据，还要路由其他网络协议的数据，甚至还要在不同网络协议之间做转换。而如果把把这些协议都做到SAN交换机上，那么这台SAN交换机，就是一台不折不扣的SAN路由器了。对于没有费用购买SAN路由器的用户来说，用一个层层协议转换设备来完成也是很划算的，这就像给照相机加一层层的特殊镜头一样。图17-7是一个层层协议转换器的实例。

图17-7 利用SDH网络连接后端网络

源端的纯FC协议，经过FCIP网关，变成了基于以太网的IP协议（FC over IP over ETH），经过E1/以太网转换器，承载到了E1协议之上，然后多路E1信号汇聚到光端机，通过一条或者几条光纤，传输给电信部门的SDH交换设施上进行传输，到达目的之后，进行相反的动作，最终转换成纯FC协议。这样，源和目的都不会感觉到中间一层层协议转换设备的存在。

主站点和备份站点的后端SAN交换机能够成功连接之后，两个SAN网络便可以融合了，就像一个SAN网络一样。所以，主站点的服务器也就可以访问到备份站点的磁盘阵列。这样，不需要经过前端网络，就可以直接访问备份站点的存储设备，也就可以直接在备份站点的存储设备上读写数据了。如图17-8所示，备份站点磁盘阵列上的一个LUN（卷

B) 可以直接被主站点的服务器识别，这样，主站点的服务器就可以同时操作本地磁盘阵列和备份站点的磁盘阵列了。

图17-8 后端网络通路示意图

我们来看一下这种方式下数据走过的路径：

本地磁盘阵列 → SAN网络交换设施 → 本地服务器内存 → SAN网络交换设施 → 通过协转流入电信部门网络（如果有） → 远端SAN网络交换设施 → 远端磁盘阵列，如图17-9所示。

图17-9 经过后端网络同步数据

分析： 上述的两种方式中，第二种方式的步骤比第一种少了两步，数据到达远端SAN交换设施之后，立即被传送到了磁盘阵列这个最终目标，而不必再经过一台服务器了，为何呢？

因为第二种方式中，主站点的服务器对备份站点的存储设备有了直接访问权，而第一种方式中，双方都没有对方存储设备的直接访问权，必须通过对方服务器的参与。

然而，第二种方式中数据仍然至少需要经过一台服务器，为何呢？因为涡轮泵（实现数据同步功能的软件）是运行在服务器上的，没有涡轮泵，水就不会流动，数据也无法流动，而不可能有一种泵，可以让水不经过它就可以流动。同样，数据流也不可能不经过泵就自己流动。

现在，两个SAN已经连接了，而且主站点的服务器可以畅通无阻地访问备份站点的磁盘阵列存储设备了，万事俱备，只欠东风。究竟在这种方式中，要怎么来设计这个涡轮泵呢？

其实没有什么特别之处，我们来分析第一种方式的同步方法，那个泵用的是从本地提取数据，发送到前端网络，网络那头用一个接收者将接收的数据写入到盘阵中。

分析：同理，第二种方式下，大思路当然还是这样，只不过是从后端网络提取数据之后，再发送回后端网络的另一个目的，然而这一切只需要一个泵就能完成，因为这个泵现在可以掌管数据的起源设备和数据的终结设备了。

我们理所当然地设计了这个泵，它的作用方式就是，将数据从本地的卷A中提取出来，然后直接通过SAN网络写入位于备份站点的卷B。如果数据是直接在内存中生成的，需要写入保存，则写入本地卷A一份，同时写入远端的卷B一份。这种方式显然比第一种方式来得快，但是它对网络速度要求更高，成本也更高。第一种方式中，有两个泵，而第二种方式中，只有一个泵，这样会不会造成“动力”不足呢？不会的，水在流动过程中是有阻力的，而数据流是没有阻力的，所以如果增加额外的泵，反而会影响数据流的速度。

这种实现方式又叫做“卷镜像”，意思就是两个卷像镜物与实物完全一样。第一种方式为何不叫镜像呢？因为第一种方式跨越的距离太远，这样不能达到两个卷在任何时刻的数据都相同，在讲同步和异步的时候还会涉及这方面的问题。这种方式能很好地保证数据同步的实时性，但是不适合远距离大数据量数据同步，除非不惜成本搭建高速远距离专线链路。

第二种方式，卷同步软件是工作在卷这一层的，所以它检测的是数据块的变化而不是文件的变化，同步的数据内容是数据块而不是文件，和第一种方式有所不同。

1. Veritas Volume Manager软件介绍

Veritas Volume Manager也是Veritas公司Storage Foundation套件中的一个模块，它的功能就是辅助或者代替操作系统自己的磁盘管理模块来管理底层的物理磁盘（当然也有可能是SAN上的LUN逻辑磁盘）。

一般操作系统自己的磁盘管理模块功能有限，比如Windows提供的磁盘管理器组件，其功能只限于对识别到的物理磁盘进行分区、格式化、挂载到某个盘符下，并且分区只能连续，而且不能动态调整分区大小，要调整也只能删除分区再重新建立。当然一些第三方软件可以做到调整分区大小，不过仍然需要重启。Windows Server操作系统提供的动态磁盘管理，虽然可以做到RAID卡的部分功能，但是仍然不够灵活，而且效率低下。而VxVM（Veritas Volume Manager）卷管理软件可以彻底替换操作系统的卷管理功能。

提示： 本书第5章中曾经通俗地阐释了卷管理软件的思想，VxVM就是这样一个卷管理软件，它把操作系统底层的磁盘系统当作“面团”，可以揉合起来，然后再分配。

它改变了操作系统的磁盘管理器的分区管理的闲置，将所有磁盘虚拟成卷池，然后从池中分配新的卷，新卷可以动态地增大和减小容量，可以动态分割、合并。支持卷多重镜像。支持RAID 0，RAID 1，RAID 0+1，RAID 5。支持RAID组在线动态扩容，可随时向RAID组中添加新磁盘而不影响使用。卷之上还需要有一层文件系统，VxVM同样有自己的文件系统，叫做VxFS。VxFS是一个高效的日志型文件系统，这就使得在发生崩溃之后文件系统的自检过程非常快。另外，还支持文件系统大小动态扩充和收缩。支持Direct IO。支持文件系统快照功能。

用户只要在服务器上安装VxVM软件，经过相关配置，就可以实现对服务器上两个卷的镜像操作，实现两个卷的数据同步。一旦某时刻主站点发生故障，则备份站点的卷上数据和主站点发生故障的时候完全一致。此时只要在备份站点的服务器上挂载这个卷到某个盘符（Windows）或者目录（UNIX）下，便可以继续使用了。

2. Logcal Volume Manager软件介绍

Logcal Volume Manager（LVM）是Linux系统上的一个开源的软件，后来被IBM的AIX操作系统用于默认的卷管理模块。LVM相对于VxVM来说，是一个更加开放、通用的卷管理软件。LVM同样也可以对两个卷进行镜像操作。在本书第5章已经介绍过LVM，这里就不做过多描述了。

17.2.4 通过数据存储设备软件实现专用网络同步

在前两种方式中，描述过数据要流动，就需要一个泵来提供动力。第一种方式中，有两个泵，数据流经的管道最长；第二种方式中，有一个泵，数据流经的管道比第一种要短。这两种方式，泵都被安装在了服务器上。而第三种方式，泵没有安装在服务器上，也没有安装在网络设备上，而是被安装在了存储设备上，如图17-10所示。

图17-10 经过后端网络同步数据

数据最终还是要存储在存储设备上，与其让别人从自己身上提取数据然后发送到远端，不如自己动手，丰衣足食。第三种方式就是利用的这种思想，自己做主将自己的数据通过后端SAN网络设施传输到目标设备上。

如图所示，主站点的磁盘阵列设备上的同步软件，从自身的一个卷（LUN A）提取数据，通过SAN交换机传输给了备份站点的磁盘阵列设备上的同步软件接收端，并将接收到的数据写入镜像卷（LUN B）。

数据流的路径如下：本地磁盘阵列→本地SAN网络交换设施→电信部门交换机组（如果有）→远端SAN网络交换设施→远端磁盘阵列。

路径比第二种方式又少了两步。更加重要的是，这种方式彻底解脱了服务器，服务器上不需要增加任何额外的负担，所有工作全部由磁盘阵列设备自己完成。

提示： 在本书第5章详细讲解过磁盘阵列。磁盘阵列本身就是一个计算机系统，有自己的CPU、RAM、ROM，甚至有自己的磁盘。磁盘阵列就是一台管理和虚拟化大量物理磁盘的主机系统。既然是主机系统，那当然可以在其上运行各种功能的软件了。所以，这种数据同步软件应运而生。

目前几乎每个厂家的高端磁盘阵列设备，都具有数据同步功能。比如IBM公司DS系列盘阵上的数据同步功能叫做Remote Mirror，HDS公司的叫做TrueCopy，EMC公司的叫做SRDF。不管叫什么，它们的思想和原理都是一样的，只不过在实现方法和效果上有所不同。

这种方式的数据同步，由于底层存储设备不会识别卷上的文件系统，所以同步的是块而不是文件，也就是说存储系统只要发现某卷上的某个块变化了，就会把这个块复制到远程设备上。此外，备份站点的存储设备必须和主站点的存储设备型号一致，因为不同厂家的磁盘阵列产品之间无法做数据同步。而在主机上的同步引擎，就没有这种限制，因为主机上的同步软件所操作的是操作系统卷，而不是磁盘阵列上的卷，

操作系统隐藏了底层存储阵列上的卷。不管什么厂家什么型号的盘阵，经过了操作系统的屏蔽，对应用程序看来，统统都是一个卷，或者一个盘符或目录。

17.2.5 案例：IBM公司Remote Mirror容灾实施

Remote Mirror是IBM公司的DS4000系列中端磁盘阵列上的一个软件模块，其功能就是将本地磁盘阵列上某个或者某些卷的数据，同步到远端磁盘阵列上对应的卷，支持同步和异步复制，支持一致性组。

某企业存储系统如图17-11所示。

图17-11 某企业存储系统图

这个企业有两台DS4500磁盘阵列，主、备份站点各一台。现在将主站点的两个卷LUN A和LUN B同步到备份站点的两个卷LUN C和LUN D上。此时需要启动DS4500的Remote Mirror功能。启用这个功能，要求主站点和备份站点上必须分别建立一个用于数据缓冲以及相关重要数据存放的卷，且必须为镜像卷，大小100MB即可。

(1) 在Storage Manager配置界面中，选择Storage Subsystem → remote mirror → Active菜单命令，选择在现有的array上建立一个mirror的Repository逻辑卷，如图17-12所示。

图17-12 创建Repository卷

(2) 单击Next按钮，弹出如图17-13所示的对话框，提示DS4500磁盘阵列将前端的2号端口专门用于连接远程的磁盘阵列设备来传输数据，所以这个端口将禁止用于其他主机的连接。

图17-13 重要提示窗口

(3) 单击**Finish**按钮，提示在备份节点上也需要相同的操作，如图17-14所示。

图17-14 提示备份节点需要相同的操作

(4) 主站点上的两个**Repository**建立成功后的界面，如图17-15所示。

图17-15 成功建立Repository卷

(5) 在备份站点上重复上述步骤。然后，在备份站点磁盘阵列上建立两个目标卷，大小必须大于源卷，如图17-16所示。

图17-16 建立两个目标卷

(6) 在主节点上右击**Create Remote Mirror**，如图17-17和图17-18所示。

图17-17 创建远程镜像 图17-18 创建远程镜像

(7) 此时，本地磁盘阵列会显示出网络上的另一台磁盘阵列设备，如图17-19所示。

图17-19 远程设备列表

(8) 选择备份磁盘阵列上的镜像目标盘，如图17-20所示。

图17-20 远程设备上的卷列表

(9) 选择模式，如图17-21所示。

图17-21 同步或者异步模式选择

(10) 设置同步的优先级，如图17-22所示。

图17-22 同步优先级选择

(11) 输入“yes”以便确认操作，如图17-23所示。

图17-23 确认

(12) 完成后出现如图17-24所示的提示框。

图17-24 成功提示

(13) 用同样的方法作另外一个确认提示，如图17-25所示。

图17-25 确认提示

(14) 完成后，监控主站点磁盘阵列的变化，如图17-26所示。

图17-26 镜像后卷图标的变化

(15) 监控备份站点磁盘阵列的变化，如图17-27所示。

图17-27 备份站点图标的变化

(16) 主站上右击源盘/属性可以查看镜像的完成情况，如图17-28所示。

图17-28 镜像状态窗口

(17) 备份站方法相同，如图17-29所示。

图17-29 备份站镜像状态窗口

17.2.6 小结

纵观以上三种数据同步的方式，可以发现以下特点。

第一种方式数据经过的路径最长，同步实时性最差，但是也最廉价。

第二种方式数据经过的路径适中，数据同步实时性强，但是对后端链路要求比第一种高，不适合大量数据同步。

第三种方式，数据经过的路径最短，对服务器性能没有影响，但是仍然不适合在远距离低速链路的环境下运行，而且还不能保证数据对应用程序的可用性。因为存储设备与应用程序之间还有操作系统这一层，操作系统有自己的缓存机制，如果存储设备上的数据同步引擎没有与操作系统配合良好的话，很有可能造成数据的不一致性，这样会影响到应用程序，甚至使应用程序崩溃。

主站点发生故障之后，备份站点的存储设备会感知到，然后强行接管主站点的工作，断开同步连接，备份站点成为当前的主站点，接受应用程序的读写操作，并记录自从断开连接之后发生变化的数据块。待探测到主站点恢复正常之后（或者手动重新配置同步），备份站点先将变化的数据块复制回原来的主站点，复制完成后，原来的主站点再次接管回主角色，成为当前的主站点，接受应用程序的读写操作。

17.3 容灾中数据的同步复制和异步复制

17.3.1 同步复制例解

下面分析一个实际容灾案例中数据的流动情况。这是一个基于存储设备的自主同步的环境，如图17-30所示。

图17-30 同步复制的过程

(1) 某时刻，主站点服务器向磁盘阵列发出一个IO请求，向某个LBA写入数据。待写的数据已经进入了磁盘阵列的缓存中，但是此时磁盘阵列控制器不会给服务器的SAN网络适配器驱动程序发送写入成功的应答，所以发起这个IO的应用程序也不会得到写入成功的应答。

(2) 主站点磁盘阵列将变化的数据从缓存中写入LUN A中（根据控制器策略，写入一般会有延迟）。与此同时，主站点的数据同步引擎获知到了这个变化，立即将变化的数据块从缓存中直接通过SAN交换机发往备份站点的磁盘阵列上的缓存中。

(3) 备份站点磁盘阵列上运行的数据同步引擎接收端成功地接收到数据块之后，会在底层FC协议隐式地发送一个ACK应答，或者通过上层显式地发送给主站点一个应答。

(4) 主站点接收到这个应答之后，立即向服务器发送一个FC协议的隐式ACK应答，这样，服务器上的FC HBA驱动程序便会探测到发送成功，从而一层层向操作系统的更上层发送成功信号。最终应用程序会得到这个成功的信号。

(5) 如果按照上述方式进行，即如果备份站点的磁盘阵列由于某

种原因迟迟未收到数据，则不会发送应答信号，那么主站点的磁盘阵列控制器也就不会给服务器发送写入成功的信号，这样服务器上的应用程序就会处于等待状态，造成应用程序等待，从而连接应用程序的客户端也得不到响应。如果应用程序使用的是同步IO，则其相关的进程或者线程就会被挂起。这种现象也叫做IO Wait，即IO等待，意思就是向存储设备发起一个IO而迟迟接收不到写入成功的应答信号。如果连接两个站点之间的网络链路出现拥塞、故障，便会发生IO Wait。

上述的数据复制方式，就叫做同步复制，因为主站点必须等待备份站点的成功信号，两边保持严格的同步，步调一致，一荣俱荣，一损俱损。

17.3.2 异步复制例解

再来看另外一种实现方式，如图17-31所示。

图17-31 异步复制的过程

（1）某时刻，主站点服务器向磁盘阵列发出一个IO请求，向某个LBA写入数据。待写的数据已经进入了磁盘阵列的缓存中，但是此时磁盘阵列控制器不会给服务器的SAN网络适配器驱动程序发送写入成功的应答，所以发起这个IO的应用程序也不会得到写入成功的应答。

（2）主站点磁盘阵列控制器根据策略，如果设置为Write Back模式，则在第一步之后立即向服务器发送FC协议的底层应答。如果设置为Write Through模式，则先将数据写入LUN A，然后再向服务器应答。

（3）主站点磁盘阵列将这份数据通过SAN网络发送给备份站点的磁盘阵列缓存中。

（4）备份站点磁盘阵列成功接收后，返回成功信号。

如果按照上述方式进行，即，主站点磁盘阵列只要接收到服务器写入的数据，就立即向服务器返回成功信号，这样应用程序不需要等待，数据同步动作不会影响应用程序的响应时间。向服务器发送变化的数据，可以在稍后进行，而不必严格同步。这种数据复制的方式就叫做异步复制。也就是说两边步调无须一致，保证重要的事情先完成，不重要的稍后再说。一旦遇到网络连接阻塞或者中断，只要服务器还能访问本地的磁盘阵列，那么应用就不会受丝毫影响，本地磁盘阵列会记录自从网络断开之后，本地卷上所有发生变化的数据块的位置，待网络恢复之后，本地磁盘阵列会根据这些记录，将发生变化的数据块继续复制到远程备份磁盘阵列。

有得必有失。异步复制保证了服务器应用程序的响应速度，然而付出了代价，这个代价就是牺牲了主站点和备份站点数据的严格一致。主站点的数据和备份站点的数据会有一个间隙（GAP），也就是未被成功复制到远端而积压在本地的数据。此时如果主站点一旦发生故障，这部分数据将永久丢失。而同步复制方式下，没有间隙，如果主站点发生了故障，备份站点的数据就是主站点发生故障那个时刻的严格数据镜像，不会有数据丢失。同样，同步复制的代价，就是牺牲了服务器的应用程序响应时间。

提示： 在实际容灾系统设计的时候，一定要考虑这一点，要明白用户是愿意牺牲数据安全性来换取高响应时间，还是愿意牺牲响应时间来换取数据的安全性。

目前很多设备厂商都有折中的解决方案，比如在网络正常的情况下，实现同步复制，一旦检测到网络连接超时，则转为异步复制，待网

络正常后，再转为同步复制。

17.4 容灾系统数据一致性保证与故障恢复机制

本书第16章中在介绍CDP的时候曾经提过数据一致性的问题。本节将具体分析一下数据一致性的保证办法和具体技术细节。

17.4.1 数据一致性问题的产生

设想这样一种情况，如图17-32所示，主机HOST上运行了一个数据库系统，数据库将Online Log存放在本地站点存储设备的Log卷中，数据文件存放在DAT卷中。

图17-32 数据一致性示例

（1）某时刻，数据库向Log中写入了一条Transaction记录，主机将这个针对Log卷的写请求发送给本地站点存储设备，并等待写入成功的SCSI应答。

（2）本地站点存储设备接收到了针对Log卷的写IO，在将数据写入Log卷的同时，将这份IO数据发送至远程容灾站点的存储设备。

（3）远程存储设备接收到这个针对Log卷的IO，立即将其写入缓存（或者写入磁盘，视Cache写策略而定），并通知本地站点存储设备IO完成。

（4）本地存储设备接收到远程IO完成的消息，立即通知HOST主机本次针对Log卷的写IO完成。

在主机上的数据库系统接收到Log写入成功之后，才可以将对应的Transaction数据写入数据文件，如果数据库并未接收到成功的消息，则

不会继续下一步动作。在将数据写入DAT卷的时候，系统进行与刚才的4步相同的动作。可以看到，在这种完全同步的数据复制情况下，不会产生任何数据一致性问题，写IO完全按照先后顺序被同时体现到本地和远程存储系统中，任何一步中断，下一步就不会发生。

注意： 同步IO所能严格保证的也只是灾备端数据的时序一致性，而不能保证到应用层的一致性。为何呢？因为比如数据库系统，都有自己的缓存，其中包含有大量的脏数据，只要这些脏数据不被刷盘，那么存储系统就算是用了同步IO，也不能把这些脏数据同步到远程存储中。而异步数据复制则可以使用运行在客户端的代理程序，在每次数据复制触发之前，通过代理程序将对应应用系统的脏数据刷新到存储系统中，然后再触发一份本地存储的快照，然后再触发数据复制过程，这样就可以保证应用级别的数据一致性了。

然而，在异步数据复制过程中，事情可就不是这样了，会出现这样一种情况，即先后发生的两个有逻辑关联性的时间，在被复制到远程站点之后，远程系统可能只保存了后来发生的事件，而先发生的事件却没了。这就是彻底的数据不一致，严重时会导致应用程序在分析这些数据时产生异常甚至无法成功启动，最严重的则是应用程序成功启动了，但是没有一致性检查机制，直接在这份不一致的逻辑错乱数据基础上继续运行处理，这种无知行为可能会造成更加惨痛的后果，比如程序给出了不符合实际的错误数据从而影响人类的行为。

如图17-33所示，同样为上述的拓扑，复制方式改为异步方式。

图17-33 数据一致性示例

(1) 某时刻，主机向本地存储的Log卷写入一份数据。

(2) 本地存储设备接收到这份IO数据之后，将其写入缓存，然后立即将写入成功的应答消息返回给主机。

(3) 主机数据库系统得知成功写入Log的消息，随后便做出判断将数据写入数据文件。向DAT卷发送了一份写IO数据。

(4) 本地存储系统收到这份数据之后，立即将其写入缓存并且将写入成功的消息返回给主机。

(5) 稍后，异步复制过程开始，本地存储系统由于各种原因，首先将DAT卷刚才被更新的那份数据发送到了远程存储系统。

(6) 远程存储系统接收到这份数据，将其写入缓存然后立即返回写入成功的信息给本地存储。

(7) 之后，Log卷刚才被更新的那份数据也被发送，在尚未发送完成之前，本地站点发生灾难，整个机房由于地震完全坍塌。Log卷刚被更新的数据或变为电磁波永远消失在宇宙中。

我们来看一下灾难发生之后，远程站点的数据状态。很显然，在主站点首先发生的事件（写Log）并没有被同步到远程站点，而后来发生的事件（写数据文件）却被同步到了远程站点。此时远程站点的数据影像是一份逻辑错乱的不一致影像，需要使用和处理这份数据的应用程序来对数据做一致性检查从而恢复数据的一致性。然而这个过程也是有代价的，为了保证一致性，很有可能要丢失一部分原本不应该丢失的数据。

有人在此会产生一个质疑，是的，我们这就来论述为何本地站点不

能按照顺序来发送所有的写IO数据。如果按照顺序发送，比如使用TCP协议来发送，不就可以保证发送端和接收端的数据是绝对按照顺序排列的么？当然是这样的，本地站点在传送数据所利用的协议这方面确实没有问题，底层协议绝对是遵循先后顺序的。

问题： 本地站点在将数据传送给传输协议进行传输的时候，却并不是按照写IO发生顺序的。什么？怎么可能，难道本地存储设备不知道IO发生的先后顺序么？它不会记录一下么？答对了，本地存储系统真的就不会记录这些IO的先后顺序。

17.4.2 对异步数据复制过程中一致性保证的实现方式

1. 异步数据复制实现方式及存在的一致性问题

为何本地站点不去记录写IO的先后顺序呢？答案是为了节省计算资源和空间。首先，可以在脑海中演绎一下，一个写IO进来之后，系统首先要做的就是尽快高效地将其写入对应卷对应的磁盘，而不是在缓存中排队等待被复制到远程站点。就算是系统先将这些数据写入，之后驻留在缓存中直到被传送，这样做依然也会耗费很大的缓存空间，而且这样做搞不好会积压越来越长的队列，致使系统进退两难，即使其他任务再忙也不得不先处理这些积压的队列，导致系统前端的性能大受影响。

那怎么办呢？又不记录顺序，又不让积压在缓存排队，就这么让这些数据自由地写在硬盘上石沉大海？待到要传送这些数据到远端时，我们又该怎样大海捞针呢？呵呵，说道捞针，您还真说对了，解决这个问题的办法，就是用针，什么针？指针啊！呵呵，我们是一定要记录这些IO的行踪的，绝对不能让它们溜掉，所以，在数据复制开始之前，我们就已经为整个卷创建了一份Bitmap，如图17-34所示。

图17-34 同步位图

初始Bitmap是全为0的，只要接收到一个针对源卷某地址的写IO，就立即将Bitmap中对应这个地址的位置为1，再来一个，就再置一个，已经置过位的就不再置了，反正已经被覆盖过多次了。系统就这样一直置下去，你甚至可以在系统将整个Bitmap都置成了全1之前，什么都不做，数据不被复制，这么长时间了都不作为，也不怕此时一旦发生灾难，这一整卷的变化数据都没被复制到远程。估计没人打算这么做，一旦数据复制动作被触发，则系统会不停地从Bitmap中的第1位开始向最后一位扫描，每当发现某位为1，则表明这一位对应的源卷地址上的数据尚未被传送到远程，则系统立即读出源卷对应地址的数据并传送到远程，传送成功之后，立即将这个位置为0，表明此地址数据已被传送，然后接着扫描下一位，做同样的动作。然而，具体实现的机制并不一定就得是一位一位地扫描，可以设定一个扫描窗口长度，一批一批地扫描，数据被一批一批地读出，一批一批地传送，一批一批地置0。扫描到表尾之后立即折回表头再次重新扫描。这个Bitmap是不断地被置1和置0的，数据写线程和数据复制线程，线程同时操作这个表，争抢这个表，有可能刚被置0的位接着又被置了1，也就是对应的地址又被写入了数据。

数据复制线程和数据写线程很有可能出现竞争情况，比如复制线程扫描到某个为1的位并且准备读取对应的数据进行传送，但是数据还没有被传送完之前，数据写线程收到针对这个地址的又一次写IO，则如果此时写线程再次将这一位置为1的话（原本就是1），当复制线程成功传送完数据之后，将会把这个位置为0，而这是不应该发生的，也就是说漏掉了一个写IO没有被复制到远程，这在数据一致性上是绝对不允许的。但是这里是异步复制，利用这种扫描Bitmap的方式进行的异步复制

本来就不可能保证数据的先后性，也就根本谈不上一致性，那么刚才那种竞争引发的问题是否可以忽略不计了呢？

不可以。如果最后被更新的地址的数据在很长时间内（或者永远）都不被再次更新，那么最后一次的更新就需要等待很长时间，或者永远无法传递到远程。一旦在这段时间内（最后一次漏掉更新后直到下一次这个地址再次被更改的期间），操作员希望将异步模式变为同步模式，先暂停应用程序IO，等待数据完全同步到远程之后，将复制关系改为同步，一段时间内，这个地址仍然未被更改，而此时系统发生灾难。操作员庆幸地认为两个站点的数据是完全一致的（由于之前变为了同步模式）。可惜，他错了，他被这个Bug搞惨了，由于最后那个被更新的数据块永远也没有被同步到远程，而这个数据块内恰好又是一笔非常关键的数据，异地启动应用程序之后，由于数据的丢失，给业务带来了巨大的损失。当然，这是一个极端的例子，但是也不能不防止。所以，还是引入避免竞争的设计为好，比如在发生置位冲突时，数据写线程先将保存状态，然后继续执行，等待复制线程结束传送置位之后，再重新置位。

注意： 数据写线程在接收到写IO之后一定要先将Bitmap置位成功后再将数据写入源卷，数据写入源卷后才通知上层成功完成。其中任何一步被异常中断的话，那么随后的动作都必须停止。究其原因，是因为如果先将数据写入硬盘后再置位，一旦在尚未置位成功前发生异常中断，比如系统Down机，则此时Bitmap中对应这个地址的位依然为0（如果置位前就为1则忽略本情景），而源卷对应地址却被更改过了数据，此时，再次同步时，就会发生与上文相同的Bug。而如果先置位成功再写入源卷，即便置位成功后，写入源卷之前，系统Down机，那么

重启后再次同步的过程就会将原本已经同步的数据块再同步一次，无非就是浪费了一点可以忽略不计的链路带宽，但是不会导致Bug。

在进行数据异步复制的初期，系统需要首先将源卷的所有内容复制到目标卷，然后才可以开始Catchup追踪不断变化的数据并且持续传送。这个步骤叫做初始化传输。此时，在初始化传输之前，系统会生成一份全为1的Bitmap，这样，扫描线程就需要传输源卷上的所有数据到远程了，其他后续步骤与上文所述的后期Catchup过程相同，唯一区别就是初始化传输时需要Catchup的是整个卷。

这种数据异步复制设计导致数据传输和数据写入的同步时间方面总有一段Gap，扫描线程在不断追赶不断被写入的IO以将它们传输至远端，受制于系统写IO负载和数据传输线路的因素，很少有两者齐头并进的时候。所以异步数据容灾在主站点灾难发生的时候几乎都是要丢数据的。

由于针对源卷的写IO地址是随机的乱序的，而系统扫描Bitmap时不能感知IO的顺序，只能按照自然数顺序扫描以便传送被更新的数据，这样做的结果就是，数据被传送到远程站点的时间也不是按照写IO发生的先后顺序排列的，对端接收和写入的时候当然也不是按照先后顺序的，这就最终导致了容灾端存储系统中数据的不一致。

如果用这种Bitmap记录模式来进行数据异步复制，不仅仅是在多个卷之间可能发生不一致，而且就算本地和远程各只有一个卷，那依然也会发生乱序传输，也会不一致。

有人再次质疑，称其有办法做到既按照顺序传输又不影响性能。写

IO一进来，也该怎么写盘就怎么写，但是如果为每个卷也创建一张表，每个表保存每一个写IO的LBA地址和当前系统的时间戳，如果遇到一个以前已经保存过的地址条目，则删除之前的，保存最新的时间戳。这样系统到时候就可以扫描时间戳从而按照先后顺序发送数据了。

这种设计存在问题。首先，这张表以谁来排序呢？LBA地址还是时间戳？一定要以时间戳来排序，否则还需要做额外的索引表以便查询，既然这样，那么如果遇到刚才说的那种情况，即针对同一个LBA再次由写IO进入，那么就删除原来的条目，此时，既然表是以时间戳排序，那么系统此时搜索的是这个LBA针对的条目，LBA是乱序的，怎么保证查询性能？如果还需要额外做索引的话，那就得不偿失了。就算做了索引，快速查到了对应条目将其删除之后，这个空位必须要利用的，否则表将无限扩大，那么就要将整个底部上移，这些动作都是非常耗费资源的。

提示： 另外，时间戳的粒度有多大？参考本书第16章对于CDP回溯粒度的像滚论述即又可得，这种设计实际上也根本无法保证能够记录每一个IO的先后顺序。所以，这种设计太过复杂而且还不能百分百保证顺序，因此不能应用。

通过快照来做异步复制

首先在源卷上做快照，然后将这份快照对应的数据块全部复制到远程站点。在复制期间，源卷数据一定会有后续的更改，不管它，因为会被CoFW，还有一份快照留底。等首次复制完成之后，目标卷做一次快照，这份快照内容上等同于源卷的第一次快照。一段间隔之后，源卷再次做一份快照，然后比对本次源卷快照和上一次源卷快照之间的变化的数据块，比方法见其他章节。得到变化数据块列表之后，将这些数据

块依次复制到目标卷，复制完成后目标卷做一次快照，然后将上一个目标卷快照删除，同时，源卷也将第一次快照删除，两边各自都留存最近这一次快照，然后一段间隔之后，重复相同的过程。

这种做法的好处是可以充分利用原有的快照功能，无须做过多开发即可实现远程复制功能，但是做不到同步复制，而且快照的间隔不能过短，所以RPO会加长。但是不失为一种简便的方法，同时这种方法在实现一致性保护方面也比较容易，比如做快照时如果引入主机端一致性代理，则可以做到应用层的一致性而不仅是底层时序一致性了。

2. 数据一致性保证的办法

上文中描述了利用追踪位图法来进行异步复制过程中所存在的时序一致性无法保证的问题。本节介绍三种用来保证异步复制过程中数据一致性的设计思想和方法。

1) 设计思想1：基于追踪日志法

最简单的一种保证数据时序一致性的方法，就是将所有进入的写IO写入源数据卷的同时，将其复制一份暂存在一个日志中，完全按照IO进入的先后顺序排列。之后，按照FIFO先进先出的方式将日志进行回放，生成数据流通过网络传输到远程阵列中。这种方法最彻底地保证了时序一致性，但是存在的问题也不可忽略，也就是这种日志需要占据大量空间，如果对于一个写负载很重的系统来讲，日志将占用大量空间。此外，写日志过程必须发生在IO路径前部，因为如果不在前部将IO顺序记录下来，那么一旦等到写IO全部进入后端Page Cache缓存后，将无法再分辨出顺序。所以，这种做法对IO资源、计算资源、存储空间资源都有很大影响。相比于追踪位图法来讲，不适合高负载的环境下。

2) 设计思想2：基于追踪位图的批量传输法

(1) 单本地存储系统解决办法

通过前文中对追踪位图法的分析看到，时序一致性难以保证。事已至此，是否这个问题就无解了呢？难道只有同步数据复制才能保证一致性么？

非也。人的智慧是无穷的。要解决这个问题，必须找到一个能够分辨单个IO顺序的机制或者位置，而且不能用时间戳，因为已经讨论过时间戳的局限性了。而整个存储系统唯一能够辨别IO先后顺序的地方就是总入口的IO Queue。Queue是一个先进先出的队列，虽然队列不对每个IO记录时间戳，但是如果在Queue中任何一点切开暂停，或者叫Suspend，就一定能够保证切口两侧的IO一定是按照时间先后排列的。

此时，如果系统首先将所有Suspend之前发生的IO统统复制到远程站点而确保本地站点在传送这些数据的过程中不发生灾难或链路问题，也就是说必须保证这些Suspend之前发生的所有写IO无一遗漏地被复制到远程，那么也就保证了远程站点数据的一致性。如果复制过程中发生灾难或者链路问题，即数据只传输了一半，而本地系统在传输这些数据的时候是乱序传输的，并不能分辨被Suspend之前发生的这些IO的先后顺序，所以远程系统中的那一半数据，很有可能就是不一致的。这个问题的解决办法我们在下文中详细论述，在此我们先一路假设灾难和链路问题都不会发生。Suspend之前所发生的IO，其处理过程与上一节中所述的异步复制过程相同，系统也同样需要在Bitmap中标记对应的位，再等待扫描线程的扫描然后传送。

Suspend之前发生的IO在被复制到远程的过程中，被Suspend住的那

些写IO该如何处理呢？此时系统绝不可能一直将它们Suspend，因为异步复制是需要时间的，而主机客户端的IO决不能为了等待它们完成而一直被Suspend住。实际上，当系统做出Suspend动作之后，会将被掐住的部分另存为一个新写IO Queue并且立即开始处理，漏下去的那部分Queue也同时会被系统逐渐处理掉。然后系统再从新Queue中再次执行Suspend——传输过程，周而复始。

新Queue与原Queue不能共用一份Bitmap（前一节所述的Bitmap），因为Suspend之后系统将要传输的只是被漏下去的那部分原Queue，只有这部分IO才会走前一节所述的异步复制流程，更新Bitmap从而等待被扫描后传送。如果新Queue中的IO此时插进来，那IO的顺序就无从保证了，所以需要为新Queue也建立一个对应的空的全0的Bitmap，同时在处理新Queue中的IO的过程中更新这份新Bitmap。

当系统将Suspend之前的所有IO复制到远程的时候，被复制出去的IO在旧Bitmap中对应的位并不被重置为0，目的是为了防止复制过程被突然中断。如果一旦被中断，远程系统就需要将数据回退到复制之前的状态（下文详述），而且本地系统的旧Bitmap中的所有信息都需要保留，并与新Bitmap进行OR操作从而合并为一份活动的Bitmap，然后等待下次Suspend的到来，即本次复制宣告失败。如果复制过程成功结束，则旧活动Bitmap无须保留，系统会将其删除，然后将新Bitmap角色转变为活动Bitmap，并创建一份新的空Bitmap以供下次Suspend时使用。

整个过程的示意图如图17-35所示。

图17-35 批量传送思想

就这样，系统在执行完一次批量传输后，接着再次Suspend，重复刚才的动作，周而复始地将本地存储系统发生的写IO数据一批一批地传送至远程站点。所以，怎么折腾也是异步，没必要非得一个一个地按照顺序传送IO，一批一批地传既简单又划算。

这个过程的实质其实就是批与批之间可以保证绝对按照时间顺序排列，但是一批当中的IO则无法判断先后时间。所以上文中提到过，在批量传输过程中一定要保证本地不发生灾难或者链路没有问题。

还没结束。细心的人已经发现了。上面的做法不正是刻舟求剑、掩耳盗铃么？虽然顶上给掐住一下子，形成两份bitmap，但是底下的卷就一个，数据不停地被更新到卷中的数据块中，而当前活动bitmap是不允许再更改的，被冻住了，本批次需要传输哪些块已经定死了，而这也正是当初为何要新生成一份第二bitmap的原因。但是，当前活动bitmap冻住了，底层卷却没有被冻住，所有更新都是直接覆盖源数据块的。试想这样一种情况：某时刻系统扫描活动位图进行批量同步，当扫描到一半的时候，底层卷有某个块被更新了，而这个块地址恰好落在已扫描完成的区域，那么此时没有任何问题。但是后来，底层卷中某个块又被更新了，而这个块地址恰好落入了尚未被扫描到的区域，那么当系统扫描到活动bitmap中对应这个块的位的时候，如果这个位恰好又为1，那么就会将对应的块读出来并且传送到远程。这样的话远端的数据卷的状态就时序不一致了，因为这个块是后来发生的，在它之前还有一个块的更新，而这笔更新却并没有传到远程，这就是典型的不一致。如何解决？两条路：要么将底层卷的分成两个，要么就将底层卷冻住，等本批次同步完成之后再解冻。

显然，把卷也分成两个的路子行不通，动一动内存里的数据怎么都行，但是不要去动底层的数据结构，否则会很难收场。那么就只能将底

层卷冻住了，显然想到利用快照技术来冻住卷，形成同步开始时刻的一份快照，这样的话，系统扫描活动bitmap进行传输的时候就不需要担心底层卷数据随时变化而导致的 inconsistencies。然而，也没有必要完全使用传统的快照技术，可以在其上做一些修改，删掉一些不必要的功能模块，只保留最核心的CoFW线程核心即可，并且其作用逻辑也可以更加简化：

- 只CoFW那些当前活动bitmap中被标记为1的块；
- 只CoFW那些当前活动bitmap中尚未被扫描到，也就是尚未被传送到远端的块。

当本批次传输完成之后，删掉这个快照。当下一批次同步更新开始时，再做快照，然后利用两份循环位图完成数据传输。当然，如果为了节省开发成本等因素考虑，直接使用已有的快照模块也可以。

（2）多本地存储系统解决办法

以上的设计虽然可以严格保证单台本地存储系统向单台远程存储系统复制数据时的数据一致性，但如果本地有多台物理上独立分开的存储设备，而某主机上的某个应用程序又同时向每个设备上的一个卷写数据，这些卷之间又有逻辑相关性，则此时就必须要求所有存储系统都要伺机Suspend住IO，而且Suspend的时间点也需要恰到好处，不允许某个设备上有某个IO是先于其他某设备的某IO执行的却被Suspend住，而其他设备上后发生的IO却没被Suspend住而漏下去执行了。

这种目的应该怎样达到呢？让所有系统在其Queue的任意位置Suspend？不可能，系统与系统设计不同，Queue也不同，相关联的IO处于各个系统中的Queue的位置也不同，所以系统不可能恰到好处都默契

地Suspend在一个一致点。本质上，存储系统也不可能感知其他系统中某IO与自身系统中的某IO是否有逻辑相关性。但是，我们可以巧妙地避开这个问题，从一种根本的并且自然形成的角度来解决这个问题。

设想一下，如果是从Queue的中部某处来Suspend，则多个独立存储系统一定有机制来相互协作，开发这种机制不是不可以，而是得不偿失，开销太大，有没有一种不用额外开发就自然存在的机制呢？然也。如果不是从Queue中部来Suspend，而是所有设备由其中一台设备作为Commander，向所有存储系统发出命令，令其在某时刻准点同时（关于同时，请参考下文论述）直接从各自Queue的尾部触发Suspend，即任何尚未到达存储设备内部的、主机尚未发送的，或者在主机的HBA卡Queue中的，或者已经发送但是依然在线路上传递着的所有IO，都被物理地隔离到了所有存储系统之外，这样的话，所有存储系统就都可以完全保证漏下去等待执行的所有IO中的最晚发生的一个，与被掐住的那个点最早到来的IO是先后连续的了。触发Suspend之后，按照与上一节中相同的步骤进行。同样，期间也不能出现本地灾难或者链路问题。当所有本地存储系统完成本批次的数据复制之后，会向Commander通知状态，Commander确认所有成员都完成任务之后，再次发起下一轮Suspend。如图17-36为这种思想的示意图。

图17-36 多本地存储系统解决办法

注意：这种方法要求所有本地存储设备的系统时间严格保持精确同步，否则同样会发生不一致。比如，主机分别先后于T1、T2、T3时刻向存储系统A、B、C发起三个写IO操作，现在系统决定在T4时刻发起全局Suspend，但是存储系统B的系统时间比A和C的快，则就有可能在A和C尚未到达T4时刻之前，B已经达到了规定的时间，则B此时立即Suspend，此时主机针

对B系统的写IO恰好尚未到达B，被排除在外。而晚些时刻，A和C同时达到了T4时刻，同时Suspend，而恰好针对A和C系统的写IO都已经进入了Queue，则整体系统在Suspend之前Queue中的写IO只包含了针对A和C的，没有针对B的，而针对C的写IO是晚于B发生的，所以此时整体系统数据处于不一致状态。

说到这里，又有了一个矛盾，上文曾经论述过，时间戳并不能保证单个IO级别的顺序，即使本地的所有存储系统的时间再精确，也不可能无限精确下去，总会在某个数量级产生差异，如果此时系统的写IOPS非常大，大过了时间戳的数量级，则上面的方法也就不能保证单个IO级别的顺序的Suspend粒度了。

有人出主意了，这样行不行，即触发Suspend的时候直接发送触发指令，而不是事先通知所有系统在某个时刻触发？不行的。想出这种方法的人忽略了其本质问题，即多个独立系统的时间不可无限精确，就算是用指令立即触发，那也要考虑指令到达各个独立存储系统的时延误差，和各个存储系统的处理时延，一旦处理不同步，一样还是造成了不一致。

江郎才尽了么？尚早。既然多个系统时间不可无限精确，那么单个系统的时间至少对于自己来讲是精确的，不存在不同步的问题。如果能让一个独立的系统来总控所有IO的进出，从物理上将所有IO串行化，然后在这个关口实现Suspend，那么这个问题就非常完美地解决了。看看系统拓扑图，最佳的串行化关口是谁呢？没错，就是主机。在主机上进行IO物理串行化是最彻底的。在主机的HBA驱动上层设计一个串行化器也可以说是Suspender，由这个Suspender来Suspender主机上的IO，然后通知所有其连接的本地存储系统做好准备。这一过程将会是很快的，在几十毫秒级别。随后Suspender释放主机IO。

经过这样设计之后，各个本地存储系统之间就不需要太过精确的时间同步了，几毫秒甚至几秒的误差都可以容忍，因为此时主机上的Suspend已经完全暂定了向存储系统发送IO，存储系统有足够的时间将全部剩余的IO收纳进来然后做好准备。

如图17-37所示为主机端Suspend解决办法示意图。

图17-37 主机端Suspend解决办法

虽然可以自己开发一个Suspend，但是如果有现成的东西，为何不拿来用呢？第16章介绍过VSS服务，VSS服务是一种最好的Suspend，它直接作用于应用层，Suspend在应用层，这比在哪一层Suspend都要更佳。所以，完全可以利用VSS来实现一致性组的功能。

然而，问题并未就此彻底解决。如果有多台主机共同连接本地存储系统，或者多台主机之间具有逻辑相关性，那么如果只在其中一台主机上实行Suspend，那么未必能够同时保证其他主机写IO的一致性，所以这里又回到了刚才那个问题，即如何将所有主机的系统时间做到精确同步从而精确同时Suspend，答案一样是不可能。这次是真的江郎才尽了么？依然尚早。既然主机和存储都不能做到，那为何不在它们之间插入一个独立的串行化器呢？完全可以的。交换机就是这个角色的最佳候选者，如果在交换机上开发一种模块专门实现这个动作，并且可以与阵列进行指令交互，形成一个标准协议，那就是最好不过的了。其次，虚拟化存储网关设备也同样是一个极佳的角色。

然而，目前针对多主机间一致性组实现方面，尚未有成型的产品出现。

如图17-38和图17-39所示为基于交换机和基于虚拟化网关的一致性

组实现方式。

图17-38 基于FC交换机的一致性组解决办法

图17-39 基于虚拟化网关的一致性组解决办法

提示： 目前就作者的了解程度范围内，很多人对数据一致性保护方面的理解很局限，总认为只要为每个IO打入时间戳，就一定能够保证数据是按照先后顺序写入的。这一点很具有误导性。我们可以分析一下，比如本地先后发生了A、B、C三个IO，此时给它们分别打入时间戳T1、T2、T3，然后通过链路发送到对端，然而，在发送的过程中，A先被发出，而C第二个被发出，最后一个发出的是B（根据上文所述的异步复制的设计思想，发生这种情况是肯定的），对端会分别先后接收到A、C、B。试问，此时远程系统如何判断到底什么时候将这些数据写入呢？如果系统打算在接收到A、C之后就写入，那么根据时间戳判断，C在A后面，的确，所以系统写入了A和C，殊不知，还有一个B尚未被接收到呢。如果B尚未收到之前链路断开或者本地灾难，那么远程对应的数据就是不一致的了。有人说了，远程就不能等B来了再写入么？不能。因为远程怎么知道还有一个B没过来？有人又说了，根据上文所述的设计思想，本地会一批一批地将数据传送啊，这一批中只要有一个IO没传成功，那么远程就会回滚啊！是啊，正是因为这样，那么是否还有必要给每个IO打入时间戳呢？打入了又有什么用呢？远程必须成批的数据一次写入磁盘才可以保持一致，那又何必在乎这一批IO中谁先被写入谁后被写入呢？有人又问了，我如果在本地系统发送IO的时候严格按照顺序发送，那么远程接收到数据包之后，上层的协议，比如TCP或者

FC，也一定会按照顺序排列起来，这样不就可以保证每个IO都是按照顺序进入远程系统的么？的确，是这样的。但是上文中也说过，在本地系统内记录IO的先后顺序而且还要按照顺序发送IO，这种开销太大。再说了，如果是多系统协作进行一致性保护，你还这么设计么？单系统已经够复杂，多系统你再去横向比对多个独立系统之间的IO谁先谁后的话，那就得不偿失了。

经过以上的分析，我们可以判断，给每个IO打入时间戳是根本没必要的，多此一举。如果选择使用对IO打入先后顺序信息的一致性保障设计方法的话，请使用自然数序号而不是时间戳进行。

3) 设计思想3: IO序号法

(1) 单本地存储系统解决办法

上文论述了为何时间戳无法判断单个IO的先后顺序以及不能判断两个IO之间是否还有漏掉的IO。如果要从为每一个IO打入一个信息以用来识别先后的角度解决这个问题，就需要找一种能够反映单个IO先后顺序并且还可以让接收方判断出两个识别信息之间是否还有其他识别信息未收到。很显然，这种识别信息就是自然数。

本地系统在接收到每个写IO之后，为这个IO打入一个自然数序号，在初始化数据同步完成之后所接收到的第一个IO，为其打入自然数1，然后没接收一个再+1，一直排列下去。依然使用Bitmap来对本地卷更新过的地址做记录，不同的是，这个Bitmap中还需要对应每一位增加一个对序号的记录项，当本地系统读出待传送的数据后，根据Bitmap中对应位所记录的序号，将这个序号追加在所传送的IO数据之后。远程接收

到数据之后，会按照自然数的顺序将数据写入远程卷，比如某时刻收到序号为1024的IO，将其写入卷中，随后又接收到了序号为1026的IO，此时系统就会得知序号为1025的IO丢失或者尚未传送到，系统会缓存所有已经接收到的1025之后的IO数据，直到接收到1025号IO，然后将这些数据写入卷中。

经过这样的设计，本地系统不再需要Suspend操作了，而且灾难之后丢失的数据也较上一种实现方式降低了许多（上一种一丢一整批，也就是一整份Bitmap中的数据，这种则是只丢失Gap内的数据），RPO能够降低一些。但随之而来的则是Bitmap的庞大、占用内存过多、IO数据带序号传送耗费了更多的网络带宽、定长的序号一旦耗尽需要两端协商重置序号等问题。而其中最为突出的就是耗费更多的网络带宽，异步数据复制的起因就是因为链路带宽过低不能满足同步复制的要求。而这种设计方式便是雪上加霜了，造成复制速度降低，所以，这种设计思想虽然在系统写IO负载不高的时候会降低一些RPO。但是在写负载非常高的时候，由于链路带宽的额外负担，则数据复制的Gap会加大，此时是否能降低RPO也就要综合考虑了，搞不好适得其反。

综上所述，现在大部分厂商都在使用第一种设计思想。

（2）多本地存储系统解决办法

多本地存储系统情况下，对IO序号法设计思想来说就是一个梦魇，因为此时IO序号需要在全局下保持各自的先后，而这种先后顺序仅靠每个存储系统是无法判断的。需要在一个上游设备中将IO串行化然后编号，发送给存储系统。这样做就需要改变存储系统前端，这个工程无疑是巨大的，而且也得不偿失。

所以，IO序号法目前来说尚无实际应用。本书以后的例子都基于第一种设计思想来论述。

（3）数据复制过程中的错误恢复机制

上文中留了一个问题尚未解决，即一旦在数据复制的过程中本地发生灾难或者链路出现问题中断，那么此时处于远程存储系统中的数据几乎就是不一致的。这个问题如何解决？很显然我们需要回退已经被复制到远程站点的数据，存储系统的回退，我们立即想到了快照。是的，如果在每次数据开始复制之前，能够为远程存储系统对应的卷制作一份快照的话，那么一旦后来的复制发生问题，就可以直接将对应的卷回滚到上一次快照的时间点，虽然此时丢失了数据，但是这是异步容灾所无法避免的，起码这样做能够保证数据的一致性。所以，我们在每次数据被成功地复制之后，还需要多加一个步骤，即为远程存储系统制作一份快照，制作成功后才会通知Suspender。

对于多本地存储系统的关联性数据复制，在Suspender发起Suspend指令并命令所有存储系统开始数据复制之后，一旦其中某一台存储系统发生数据复制错误，或者干脆这台设备发生故障，那么Suspender永远也不会收到这个设备返回的成功消息了，在一定的允许时间之内，Suspender只要没有从任何一台设备接收到成功消息，则便会立即向对应的所有远程存储系统设备发送命令让其回滚到上一个快照时刻。而由于本次复制并未成功，所以本地所有存储系统会将当前活动Bitmap与新Bitmap（即对应Suspend之后发生的IO的Bitmap）进行OR操作，合并为一份活动Bitmap，并创建一份新的空Bitmap以便为下次Suspend之后作为新Suspend Bitmap使用。下次Suspend之后，即使上次传了一半的数据也依然需要再传一遍，这也是为何要进行OR操作的原因。但是EMC公司SRDF/A的设计则可以在本地记录上次传过的数据，恢复之后，只传

递上次未传过的和新一轮Suspend后改变的。当然这样做就需要远程系统暂时不回滚，保持上次已经传输完成的那部分数据，当确认本地站点由于某种原因无法增量传输之后再回滚。

如果所有本地存储系统的数据复制过程都成功结束，则这些存储系统会通知Suspend，然后Suspend会向所有对应的远程存储系统发送命令，即触发一份对应卷的快照。快照成功后，Suspend继续开始Suspend操作，周而复始地将数据一批一批地复制到远程，并且每次成功后制作快照，并删除倒数第三个快照（倒数第二份快照用于异常情况之后的回滚）。

对于普通异步复制或一致性异步复制，系统Down机重启之后，将会继续进行扫描Bitmap并传送对应数据的过程。在第17.4.2节中所述的两个潜在的Bug一定要特别注意。

对于同步模式的数据复制，错误恢复过程就相对简单了。比如某时刻链路突然中断，或者系统Down机后重启。如果是严格同步，那么受影响的应用程序就会出错，业务无法继续进行。而此时可以人为干预断开同步复制逻辑Session，重新启动应用程序，对底层卷的写IO不再被同步传输到远程，但是会在本地创建一份新Bitmap用来记录所有在断开复制Session之后本地卷所有被更改过的数据地址。这样，在链路恢复之后，重新启动复制逻辑Session，此时系统就会根据这份Bitmap而只将变化过的数据再次同步到远程。以上过程可以预先在容灾系统中进行设置从而自动执行。

3. 一致性组

说到这里，该说一下一致性组了。其实仔细看完前面的论述后应该

就能体会到这个“组”字的含义了。第一个含义：数据是被一批一批，或者一组一组地传输到远程的，组和组之间能够保证先后顺序，组内则无法保证，如果一整组数据完整传到了远程，则远程的数据是一致的，如果一整组数据在传输的中途出现问题，则需要将远程的数据回滚到上一个一致点，这个目的需要用快照来完成。第二个含义：多个独立存储系统上的多个卷组成一个整体，个体与个体之间必须保证数据按照整体时间的先后传输到对方，远程的多个存储系统上的多个卷形成的组，在整体上必须保持数据一致性。

这就是一致性组技术。

4. 快照异步复制方法中的一致性保护和一致性组

基于快照的异步复制，相比上文中的异步复制设计虽然RPO加大，但是在一致性方面还是很不错的。比如在做快照时，可以通过主机端Agent来将缓存数据刷入磁盘，根据主机Agent的作用层次，最高可以做到应用级的一致性，也就是彻底的一致性保证。其次，在数据复制期间，由于两端都做了快照留底，所以不必担心中途链路中断等导致的目的端数据时序不一致状况，一旦遇到中断，则目的端回滚到最近的快照即可，随后择机继续执行数据复制。同时，在多主机多LUN多存储系统一致性组方面，更是没有问题，同一台主机上的多个LUN需要一致性组保证的话，那么可以通过主机上的Agent来同时刷这些LUN的数据，之后做快照；如果是多台主机的多个LUN（位于不同阵列），也可以通过让多台主机上的Agent同时刷对应LUN的数据，然后各自阵列做快照。这里相当于主机快照Agent充当了上文中的Suspend/Freezer了。

17.4.3 灾难后的切换与回切同步过程

容灾的目的是为了在本地发生灾难，或者并未发生破坏性灾难，但是正在遭受长时间的断电，或者通风系统、温控系统等辅助设施的长期故障，则此时可以在异地将业务重新跑起来。养兵千日，用兵一时。如果真的需要将业务切到异地运行，那么应用程序就一定需要向异地站点的卷中写数据。而一旦主站点从灾难或者长期Outage中恢复，那么就需要将业务再次回切到本地主站点执行。这个时候，就需要将在异地执行时被更改过的数据重新同步到主站点。

对于同步数据复制架构下的切换与回切，处理过程也是相当简单的。比如，主站点在任意时刻发生灾难之后，主站点和异地站点对应的数据是绝对相同的，在业务切换到异地站点执行之前，异地站点就可以对所有受影响的卷创建一份新的全0的Bitmap，用来记录所有切换之后针对本卷的写IO数据，被写过则将IO地址对应的位置为1。在主站点恢复运行之后，需要将异地站点变更的数据Resync到主站点，此时异地站点扫描每个卷的Bitmap，把为1的位所对应的地址的数据读取并覆盖到主站点对应的卷的对应地址。结束之后，应用程序即可重新启动，恢复最初的状态。

而对于普通非一致性异步复制过程，又可分为三种情况（其实这三种情况的处理方式都是相同的）。

第一种情况是：本地系统发生灾难，但是硬盘完好没被破坏。一切活动停止，数据状态永远定格，对应的本地卷Bitmap也将定格，复制到哪算哪。之后，应用程序在异地启动，启动之前，异地系统为每一个参与容灾的卷创建一份新的全0的Bitmap，然后开始接受应用的写IO，每接受一个，就在Bitmap中对应的位置1，然后将数据写入容灾卷。当主站点恢复之后，需要将在异地变化的数据重新同步Resync回主卷，此时，主站点首先将灾难之前被定格的Bitmap传送到异地，异地系统将这

份主站的Bitmap与异地的Bitmap进行OR操作（至于为何要使用OR操作，读者可以自行推导，可以参考第16章中卷Clone一节里的算法思想）。得出的新Bitmap中，为1的位就表示需要进行Resync的数据地址，然后异地系统读出对应数据传送回主站，主站覆盖之后，方可再次创建异步复制Session，系统恢复原状，之后方可启动应用程序。

第二种情况是：本地发生灾难，玉石俱焚。那么没什么好说的了，应用程序在异地启动，本地重建之后，购买全新设备，异地变为了主站点，重建后的本地变为了容灾站点，数据重新被同步。

第三种情况是：本地没有发生整体灾难，而只是对外网络链路全部中断，包括数据复制链路。此时业务需要在异地启动。之后的过程与第一种情况完全相同。

对于带一致性组的异步复制过程的错误恢复，步骤与普通异步复制过程完全相同，但是除了一点，也就是需要Resync时，本地站点需要将Suspend Queue对应的Bitmap与活动Bitmap进行OR操作后再发送给异地系统，异地系统将这份OR之后的Bitmap再次与自己的Bitmap进行OR运算之后才能得出结果。

17.4.4 周期性异步复制与连续异步复制

从上文的分析可以看出，每次异步传输触发之前，系统需要做很多处理，包括生成新位图、锁住对应的块或者生成快照等。而这些后台处理过程需要耗费一定的资源。系统可以使用两种模式来触发异步同步过程。

1. 连续异步复制

这种模式下，系统不停地做复制，本次同步完成之后，立即开始下一轮的同步，利用循环位图及快照连续不断地将数据复制到远程，而不管每次复制耗费多长时间，复制了多少数据。这种方式是一种自适应的全自动方式。

比如某时刻系统触发了复制操作，复制用了10分钟完成，这10分钟期间，源卷发生了200MB的变化，那么在历时10分钟的同步结束之后，系统立即开始对这200MB数据的继续同步；假如这次同步过程用了5分钟，而这5分钟期间源卷发生了100MB的变化，然后系统继续同步这100MB数据；用了2分钟，期间又产生了50MB的变化；再同步这50MB用了1分钟。就这样一直往下收敛，最后导致同步的间隔越来越短，每次传输的数据越来越小，最后会达到一个极端状况，即两次数据同步期间源卷没有发生改变，那么此时系统会依然执行相同过程，只是最后传输的时候发现位图都是0，所以立即完成本次传输，然后再次触发下一次传输，相当于空转，这样就耗费了不必要的资源。但是却可以最大程度地保证系统的RPO。

2. 周期性异步复制

这种模式下，系统严格按照预先设定的时间间隔或者待同步数据的积累量来触发复制，比如每15分钟，或者每积累50MB的数据量。这样做合乎常理，也节约系统资源，但是其RPO是固定的，比如就是15分钟，或者50MB的数据。灾难发生之后，这15分钟内的数据或者这50MB的数据一定是丢失找不回来的。

17.5 四大厂商的数据容灾系统方案概述

本节对四大厂商的基于存储系统底层的数据容灾方案进行简要介绍。

17.5.1 IBM公司的PPRC

PPRC其全称为Peer-to-Peer Remote Copy，即点对点的远程数据复制。它是IBM用于其DS 6000和DS 8000中高端存储平台上的一种远程数据容灾软件模块。这个名称只是一个统称，它其实包含了Metro Mirror、Global Copy和Global Mirror等多种组件。

Metro Mirror方式是一种同步数据复制方式，能够在任何情况下保证数据一致性。适用于距离较近并且链路带宽足够的两个站点之间的复制。

Global Copy方式则是一种不带一致性组功能的异步数据复制方式，这种方式不能保证数据的一致性。适用于距离较远而且链路带宽很低的情况下的数据复制。它的数据复制方式与17.4.2一节中描述的相同。

Global Mirror则是一种带有一致性组保证技术的异步数据复制方式，可以保证在单存储系统的严格数据一致性，底层也是使用双循环位图法。但是对于多存储系统，不能严格保证数据一致性，Global Copy只是使用了多存储系统时间同步，然后同一时间Suspend的模式进行数据一致性的大致保障的，如果系统IOPS超过了时间精确级别，那么依然不能保证单个IO级别的一致性。

在多存储系统架构下，用户需要指定其中一台为Master，其他参与

一致性组的系统都为Subordinate。Master负责同步全局系统时间和发送全局指令从而命令其他设备执行Suspend。当所有系统成功完成本次Suspend和数据传输之后，Master还负责向远程关联的所有存储系统发送指令，命令这些远程存储系统对相应的卷做一次Snapshot。IBM的Snapshot产品名称为“FlashCopy”，这个名字具有一定的迷惑性，中译名“闪速复制”，其实闪速的只是创建了一份地址映射表而已，数据并未被全部复制。FlashCopy在执行的时候有个选项叫做“NOCOPY”，这个选项控制着Snapshot生成后的行为，如果被设置为NOCOPY模式，则系统只保存地址映射表和被CoFW出来的数据；而如果选择了COPY模式，则系统会像Clone Split一样将共享数据块全部复制出来，加上CoFW的数据块共同在一个新的存储空间形成一份与源卷对应时刻一致的物理卷复制。

PPRC可以实现两地三中心的容灾架构。站点A与站点B处于同城距离范围内，使用Metro Mirror进行同步数据复制。站点B再与位于异地远距离范围的站点C之间进行Global Mirror带有一致性组的异步数据复制。

17.5.2 EMC公司的MirrorView、SanCopy和SRDF

1. MirrorView

MirrorView是专用于Clariion存储系统平台上的远程卷镜像软件模块。分为MirrorView/S和MirrorView/A两种模式，前者为同步镜像模式，后者为异步镜像模式。前者没什么好说的，各家实现方式都一样。MirrorView/A底层使用了双循环位图法+Snapview来保证时序一致性。

MirrorView/A只支持单存储系统中多个卷的一致性组。而IBM

PPRC则支持多独立系统中多个卷的一致性组。

2. Sancopy

Sancopy其实是一个Snapshot Replicator。也就是说，它运行于存储设备中，专门将本存储系统的某个指定的源卷的某个指定的Snapshot，直接通过后端存储网络复制到另外一台或者多台存储系统的指定的卷中。由于Sancopy是复制的快照，需要调用Snapview来实现快照的生成，所以，Sancopy的源卷只能位于Clariion平台存储系统上，Sancopy软件自身也只能安装并运行在Clariion平台上。而复制的目的卷可以位于Clariion、Symmetrix DMX以及其他经过认证的第三方存储系统中，比如IBM、HP、HDS等。支持单台存储系统的一致性组，即在Suspend后对多个源卷同时制作快照并且复制。经过第一次初始化复制之后，通过不断地对源卷制作快照并且比对前后两次快照之间的变化数据地址从而实现增量的数据传输。在初始化传输时，由于需要将源卷所有数据首先同步到目的卷，所以要求源卷在这期间不能有写IO发生。从技术角度来讲，实现允许源卷写IO的初始化传输是没有问题的，制作一份对源卷的初始快照并且传输到目的卷，此后逐渐生成快照做增量传输，慢慢达到Catchup状态，这种做法理论上讲没有问题。但是Sancopy自身无法做到，必须依附于Snapview/snap或者Snapview/Clone来做到。

Sancopy支持远程复制，支持通过iSCSI协议复制。Sancopy本质上就是把源卷所在的存储系统作为一个Initiator，目的卷所在的存储原本就是Target，这样Sancopy所在的存储系统就会识别并且挂载目的卷到本地，然后将本地源卷的某个Snapshot直接像主机访问Target一样复制到目的卷，之后再做增量复制。所以这种模式下，Target端无须做任何变化，Target此时就当Sancopy端的存储系统为主机。

正因为本质上就是Initiator-Target关系，所以Sancopy也支持将源卷某个快照复制到一个比源卷更大的目的卷。目的卷比源卷多出来的容量在物理上是追加到了源卷的尾部。

Sancopy其实与MirrorView/A在底层设计有很多都是相同的，都是基于Snapshot Catchup模式而且都需要调用Snapshot/snap。只不过Sancopy是直接以Initiator-Target模式工作，所以可以复制到众多第三方存储卷；而MirrorView需要源和目的双方都运行MirrorView的进程，不是通用的SCSI Initiator-Target模式，所以其只支持Clariion之间的互相Mirror。

3. SRDF

1) SRDF/Synchronous (SRDF/S)

这是同步数据复制，没啥好说的，全世界的厂家都是一样的实现方式。

2) SRDF/Asynchronous (SRDF/A)

这是带一致性保障的异步数据复制模式。SRDF/A的设计模式大致思想与上文介绍的无异。同样是数据批量传送，Suspend后形成一批，然后等待传输。如图17-40所示，系统Suspend之后的所有IO在图中就是Capture N，对应前文中的Suspend Queue；当前正在传输的数据对应图中的Transmit N-1；远程正在接收的数据批，对应图中的Recive N-1；远程已经接收成功的数据批，正要将其写入磁盘，对应图中的Apply N-2。

图17-40 异步SRD

注意： 为何在远程站点，数据不是一边接收一边被写入磁盘呢？因为SRDF/A没有用到任何快照技术来照下远程站点的一致性状态，与前几节的设计思想是不同的。所以，只能利用数据库所常用的Two Phase Commit思想来做，即接收到的数据批，首先写到一个额外的存储空间，确保整批数据都接收到了，然后再一次性写入磁盘。在写入的过程中，也需要再记录一个日志，因为一旦在写入过程中系统发生Down机等异常情况，重启后还可以追溯这个日志以便了解上一次成功写到哪里了，然后继续从断点处开始写。

SRDF/A与前几节介绍的实现方法还有其他一些不同，下面就追加介绍一下这些不同之处。首先，主站点复制引擎接收到针对源卷的写IO数据之后，在这些数据没有被复制到远程之前，都会驻留于内存等待传输（根据多种事实推断，SRDF/A没有利用Bitmap作为设计思想，这一点只是推断，但是无法确定）。这种设计实在不太划算，虽然能够节省一些磁盘读IO操作，但是也浪费了大量的缓存稀缺资源。

果不其然，这种设计模式最终导致其不得不考虑缓存溢出的情况，所以又开发了一些补救措施。比如，一种叫做Write Folding的技术，如果某个写IO是针对之前已经写过的地址再次写，而之前的写IO数据尚未被复制到远程，那么系统就不复制原来的写IO了转而只复制新的写IO（这一点如果利用Bitmap来实现的话是自然就可以实现的，所以再次推断SRDF/A没有用Bitmap实现）。

再严重一些，见势不妙的时候，就提高对主机写IO的响应时间以降低主机端写IO的频率（Write Pacing），其次，如果这招都不好用了，眼看缓存将满，那么还可以将缓存中的一些数据destage到磁盘，相当于page/swap，以腾出一些空间（Delta Set Extension）。这样的话，最终还

是要多耗费磁盘读了，而且是先耗费了缓存资源，可谓是赔了夫人又折兵，还给主机故意提高响应时间。其次，还有一点不同之处，这一点是个优点。即上文中曾经提到过的，在遇到传输错误、链路异常等情况时，再次传输时，SRDF/A可以只传输上次传输中未完成的部分。

以上为本地单存储环境，实现一致性组较为容易。虽然SRDF/A也是支持本地多存储系统下的一致性保障的，然而，根据推断，SRDF/A与IBM Global Mirror一样，也不能实现单个IO级别的一致性。SRDF/A与IBM Global Mirror有一点不同，即前者是将Commander放置于主机端而不是本地存储系统中的一台。让人费解的是，既然已经涉及到了主机的参与，那么为何不在主机端一并实现Serializer和Suspend的功能呢？这一点根据现有的资料判断无法判断。

至于多主机环境下的一致性组，更是无法支持了，目前似乎没有厂家支持这么高级别的一致性组保证。

17.5.3 HDS公司的Truecopy

Truecopy也分为Synchronous模式和Asynchronous模式。同步模式没什么可说的。异步模式也是使用批量传输法保证数据一致性，底层设计细节不再赘述。

17.5.4 NetApp公司的Snapmirror

Snapmirror也分为同步和异步模式。在异步模式下，Snapmirror针对源卷每隔一段时间做一次快照，通过比对两次Snapshot的变化数据地址，从而将这些变化的数据传输到远程存储系统对应的卷中。这种思想的本质其实也是批量传送，对源卷的Snapshot就相当于一次Suspend，比

对两份快照的变化数据地址，就相当于记录Bitmap。

对于多台独立存储系统之间一致性组保证方面，NetApp是基于主机VSS实现的。其Snapmanager产品（与EMC的Replication Manager类似），作为一个Requestor向VSS发送快照请求，VSS将Suspend对应的应用程序的IO，这样，不管这个应用程序需要读写的卷分布于单个存储系统还是多个存储系统，所有的系统都有足够的时间来对相应的卷进行快照操作。作为VSS的Provider，Snapdriver作为快照代理，将向所有对应的存储系统发起快照请求。快照完成后，Snapmanger可以调用Snapmirror对变化的数据进行传输。针对多主机之间的一致性组实现，尚未有解决方案。

对于同步模式，Snapmirror是利用Catchup模式实现的。Snapmirror首先进入异步模式，即数据复制的Session建立之后，立即生成首次快照，然后初次数据传递过程需要传递源卷上所有被这份Snapshot所占用的数据，需要比较长的时间。传完后再次做Snapshot，比对两次Snapshot中变化的数据地址，再次同步。就这样一直进行下去，当达到一定的Gap阈值时，系统进入同步复制阶段，针对源卷的每一个写IO在进入本地日志链后立即同步到远程，远程系统接收到写IO之后，先将其存放于Vol0下面的日志链文件中，然后在后台将日志读出重放，从而写入对应的卷。

提示： 由于WAFL的快照方式与其他厂商的常规设计不同，所以在比对两份Snapshot的时候并不是按照本书第16章中介绍的机制进行的。

WAFL的每份Snapshot其实就是一份真实的文件系统MetaData Tree。这个Tree中记录了整个物理存储空间的Bitmap，AFS以及每个

Snapshot都会有自己的Bitmap，通过下列规则比对两份Bitmap即可得出变化的数据地址。

- 规则1：后面的Bitmap中为1的但是前面Bitmap中为0的位，则表明这个地址对应的数据发生了更改。
- 规则2：后面Bitmap中为1并且前面Bitmap中也为1的位，表明这个地址对应的数据是两份Snapshot共用的，一定也没有变化过。
- 规则3：绝对不可能存在后面Bitmap中为0而前面Bitmap中为1的数据。因为WAFI从来不覆盖正在被某Snapshot占用的数据。

虽然EMC的MirrorView也同样是采用比对快照的方式进行数据复制，但是其本质上有区别。NetApp在数据复制到远程之后，快照也就自然而然地生成了，因为WAFI的快照本身就是一个卷的全部内容，而传统的Snapshot，源卷是源卷，相关的映射表和其他元数据等都存在于源卷额外的存储空间，即这些映射表里记录的是源卷的数据变化而不是记录自己所占用空间的变化，更不会将自己占用的空间的数据变更同步到远程，所以EMC的MirrorView在同步完一批数据之后需要在远程系统创建一份快照。而WAFI在同步完一批数据之后，只需要进行一个叫做“Jump Ahead”的操作即可，即，将容灾卷上的AFS入口指针指向最新的Snapshot入口指针，并且删除前一份Snapshot。当然这份Snapshot就是刚被同步过来的数据所自然形成的而不是待同步完成后制作的。

提示： 利用比对快照方式进行异步数据传送的设计模式的一个好处就是不用为错误恢复机制而额外设计复杂的流程。一旦发生Down机或者灾难，容灾站点只需要Rollback到上一个Snapshot即可，主站点重启之后立刻就可以接受IO，无须做任何前处理。关于比对Snapshot的具体技术细节实现原理请参考本书第16章的有关内容。

17.6 生产者的容灾——服务器应用程序的容灾

IT系统的生产者，也就是各种服务器上运行的应用程序。毫无疑问，主站点发生故障，必须要在备份站点重新运行这些应用程序。我们是否可以在备份站点预备应用程序的安装文件，发生故障后，在备份站点服务器上安装配置这些应用程序呢？

这么做虽然可行，但是一些较为复杂的应用程序，安装和配置要花费大量的时间，比如SAP企业REP系统的安装，可能需要一天的时间，再加上不可预料的因素，耗时可能更长。如果没有预先安装配置好这些应用程序，未雨绸缪，则事故发生的时候，企业就需要忍受停机所带来的损失了。

17.6.1 生产者容灾概述

我们必须将应用程序在备份站点预先安装并且配置好，但是不能让它们处于工作状态，应当时刻保证同一时刻只有一个站点的生产者在生产，因为IT系统生产出来的产品是具有一致性的数据，而且数据是有时效的，具有上下文联系的。IT生产是一个连续的数据处理过程，一旦中途产生数据不一致性，就需要恢复数据到某个一致的时刻，然后从这个时刻继续生产。而不像实物生产那样，产品是一件件的物品，都具有相同的属性。所以保证同一时刻，整个IT系统只有一个站点的生产者处理同一份数据，这一点非常重要。

然而，既要求两个站点同一时刻只能有一个站点的生产者处理一份数据，又要求当生产站点发生事故的时候，备份站点的生产者立即启动，接着处理备份站点经过主站点数据同步过来的数据。要做到这一

点，就需要让备份站点的应用程序感知到主站点应用程序的状态，一旦检测到主站点应用程序故障，则备份站点应用程序立即启动，开始生产。

第16章曾经说过高可用性群集，而在容灾技术领域，群集的概念扩大到了很远的范围，备份站点与主站点可能不在同一机房中而在相隔很远的两座建筑物里，甚至两个城市中。这样，备份应用程序就要跨越很远的距离与主应用程序通信来交换状态。由于应用程序运行状态数据，相对于其处理的数据来说，数据量是很小的，所以即使是跨越广域网通信，也不必担心延迟太大。如果是通过广域网连接两个站点的前端网络，则最好使用专线连接；如果是基于Internet的VPN连接，虽然可以获得高带宽/价格比，但是延迟无法保证最小，除非购买电信部门提供的QOS服务。

类似HACMP、MSCS这种HA软件，都是使用共享存储的方式来作用的，即HA系统中的所有节点，共享同一份物理存储，不管某时刻由谁来操作处理这些数据，最终的数据只有一份，而且是一致的、具有上下文逻辑关系的。而远程异地容灾系统中，数据在主站点和备份站点各有一份，而且必须保证两边数据的同步。

生产的时候，必须以一边数据为准，另一边与之同步，绝对不能发生两边同时进行生产的情况，除非两边生产者处理的是两份逻辑上无任何关联的数据。所以，远程容灾系统所要关注的有两个重要因素，即生产者和生产资料。只要生产资料在主站和备站完全同步，那么就可以逻辑上认为，数据只有一份，备站的数据是主站的镜像，平时虚无缥缈不可用；但是一旦主站发生故障，备站的镜像立即成为实实在在可用的数据，同时，生产者在备站启动生产，处理数据。这就是异地容灾。

然而，传统的基于共享存储模式的HA软件，不适用于异地容灾系统，因为共享存储模式的HA软件，是基于资源切换为基础的，它把各个组件都看成是资源，比如应用程序、IP地址、主机名、应用所要访问的存储卷等。发生故障时，备份机HA软件检测到对方的故障，然后强行将这些资源迁移到本地。比如，在备份机修改相应网卡的IP地址，并发出ARP广播来刷新所有本广域内的客户端以及本地网关设备所保存的ARP映射记录，以让所有网络上的终端获知此IP对应的新MAC地址，修改主机名映射文件（Host文件），挂载共享存储设备上的卷，最后启动备份应用系统。

应用系统可以访问已经强行挂载的共享卷而存取数据，客户端可以继续使用原来的IP地址来访问服务器上运行的应用程序，因为这个IP已经由故障的计算机转移到了备份计算机，这样，生产就可以继续进行了。要保证生产者在切换之后生产可以继续，则必须先保证生产者所依赖的所有条件已经切换成功，这些条件包括IP地址（非必需）和卷等。

1. 本地容灾系统中的两种存储模式

本地HA系统中，多个节点如果共同拥有同一个或者同几个卷，但是同一时刻只有活动节点才挂载该卷进行IO读写，这种模式就叫做共享存储模式。即HA系统中的每个节点都拥有同一份存储卷，只不过不活动的节点不对其进行挂载并IO。

如果HA系统中每个节点都有自己独占的存储卷，这些卷除了拥有者可以读写之外，任何情况下，其他节点都不能读写，数据的共享是通过同步复制技术同步到所有节点上的存储卷中的，这种方式就叫做Share-Nothing模式。即HA系统中的所有节点之间不共享任何东西，所有元素都是独享的，甚至网络地址都是各用各的。数据存在多份，每个

节点一份，节点之间通过同步复制技术来同步数据，某节点发生故障之后，这个节点对应的备份节点直接启动应用程序，由于之前数据已经在所有节点上同步，所以此时数据是完整一致的。由于Share-Nothing模式下，不存在任何的“接管”，所以此时客户端需要感知到服务端群集的这种切换动作，并通过客户端手动或者自动切换配置以便连接新服务器。表17-3对比了HA的两种存储模式。

表17-3 HA群集中两种存储模式的对比

- 数据本身是否容灾

共享存储模式下，容灾系统的各个节点共享同一份数据。如果这份数据发生损坏，则必须用备份镜像加以还原，而且需要承受停机带来的损失。而Share-Nothing模式下，系统中每个节点都有自己的数据复制，如果其中一份数据被破坏，系统可以切换到另外的节点，不影响应用，不需要停机，被损坏的数据可以在任何时候加以还原修复，并且修复后的节点可以再次加入容灾系统。

- 软硬件成本

共享存储模式下，由于各个节点需要共享一份存储数据，所以需要外接的磁盘阵列系统，而且为了保证数据访问速度，外接存储系统必须自身实现RAID机制，主机上也需要安装连接盘阵的适配器。这样就增加了整个系统的成本。Share-Nothing模式下，各个节点自身保存各自的数据，而不必使用外接存储系统。另外，共享存储模式还需要额外的HA软件及额外的成本，而Share-Nothing模式不需要。

- 前端网络资源耗费

共享存储模式下，各个节点之间交互信息一般通过以太网络，而存储数据通过后端存储网络。由于各个节点在前端网络上只传输

控制数据，所以对前端以太网络资源的耗费相对较低。而Share-Nothing模式下，由于各个节点之间的数据同步完全通过前端网络，所以对前端网络资源耗费相对较高，适合局域网环境。

- 管理难度

共享存储模式下，不但需要管理节点间的交互配置，还需要管理外部存储系统，增加了管理难度。Share-Nothing模式下，只需要管理各个节点间的交互配置即可。

- 是否需要停机

共享存储模式下，由于需要将数据从单机环境转移到共享存储环境供其他节点使用，往往需要停机来保证数据的一致性。而Share-Nothing模式下，数据同步是动态的，不需要停机。

- 实现复杂程度

首先，共享存储模式下，有三种基本元素：节点、节点间交互、共享数据。而Share-Nothing模式下，只有两种元素：节点、节点间交互。其次，如果使用共享存储模式做容灾，需要将数据移动到共享存储上，增加额外的工作量、时间和不可控因素。

- 是否需要第三方软件

共享存储模式下，备份节点需要通过第三方软件来监控主节点的状态，在发生故障的时候主动接管资源，比如各种操作系统提供的HA软件（HACMP、MSCS、SUN Cluster等）。Share-Nothing模式下不需要任何第三方软件参与。

- 故障因素数量

共享存储模式下，如果出现容灾系统本身的功能故障，需要在操作系统、应用程序、HA软件三个方面排查故障。Share-Nothing模式下，只需要在操作系统、应用程序二者之间排查故障。

2. 异地容灾系统中的IP切换

在异地容灾系统中，主服务器和备份服务器不太可能在一个广播域中，一般都是通过网关设备来转发之间通信的IP包，所以不可能用所谓资源切换的方式来切换IP地址。如果想对客户端透明，即客户端可以无须感知故障的发生，继续使用原来的IP地址来连接备份服务器，那么就需要在网络路由设备上做文章了，动态修改路由器上的路由表，将IP包路由到备份站点而不是主站点。如果客户利用域名来访问服务器，那么也可以直接在DNS设备上修改IP指向记录来完成这个功能。

最方便而且普遍的做法是：让所有客户机利用主机名来连接服务器，这样，主站点故障后，通知所有客户端修改它们的host文件即可将原来的主机名映射到新的IP地址而不用重启计算机。这方面，异地容灾系统中的HA软件几乎发挥不了作用。

3. 异地容灾系统中的卷切换

异地容灾系统中在主站点和备份站点各有卷，两个卷之间可以通过前端网络同步，或者通过后端网络同步。主站点后，备份服务器上的HA软件检测到主服务器通信失败，便会感知故障发生，然后通过某种方式，断开主卷和备份卷的同步关系（如果不断开，则卷会被锁定而不可访问）。如果同步引擎是运行在存储设备上的，那么除非HA软件可以操控运行在存储设备上的同步引擎，否则必须由系统管理员手动利用存储设备的配置工具来断开同步关系。同步关系断开后，本地的卷才能被访问，这样，HA软件才能在备份机上调用操作系统的相关功能来挂载这个卷。

如果同步引擎本身就是由运行在主机和备份机上的HA软件提供的，那么就可以实现在检测到通信失败之后，由HA软件本身来自动断开同步关系，然后在备份机上挂载对应的卷。

4. 异地容灾系统中的应用切换

应用，也就是生产者的切换，是所有HA容灾系统在故障发生后所执行的最后一步动作。与共享存储模式的HA容灾相同，异地容灾中的应用切换，也是由备份机的HA软件来执行脚本，或者通过其他功能调用相关应用的接口来启动备份机的应用。

比如，对于DB2数据库来说，启动数据库实例所使用的命令为：
db2start，HA软件只要检测到主站点故障，只要在备份机的db2cmd命令行方式下执行这条命令，便可使备份机的DB2数据库实例启动起来。应用的启动必须在所有资源成功切换到备份机后发生，因为应用启动的时候必然会读取卷上的一些数据，如果卷还没有被挂载，应用启动的时候就会报错，比如：找不到数据文件。

5. Veritas Cluster Server软件介绍

VCS（Veritas Cluster Server）可以基于VVR的配合，而实现异地容灾系统。在一个CLUSTER环境中，如果一台服务器运行多个应用，只有一个应用出现故障时，那么VCS可以只将该应用切换到预先定义的服务器上，另一个应用仍然在原来的服务器上继续运行。

VCS将其监视的应用当作一组资源来管理，这一组资源定义为资源组（RG）。例如Web-Server，要保证这个应用正常运行，VCS将监视存放数据的磁盘组，该磁盘组上的文件系统、网卡、IP地址及Web服务进程。既然VCS是基于应用的高可用软件，一台服务器上运行的多个应用可以切换到不同的服务器上。

例如，图17-41所示的服务器A运

图17-41 多Active集群

行着IIS网页访问服务和DB2数据库服务，服务器B运行着FTP服务和邮件转发服务，服务器C运行着NFS服务和SMB网络文件系统服务。当服务器A出现故障时，资源组RG-Web切换到服务器B上，资源组RG-DB2切换到服务器C上。当然条件是它们都能存取对应的应用数据。系统管理员制定合适的故障条件，例如现场完全瘫痪10分钟或某个应用停止运行半小时。当这种情况发生时，可以设定有GCM（Global Cluster Manager）自动切换应用，或向系统管理员报警，得到确认后，再切换应用。无论应用切换是自动还是需要确认，两个场地之间应用的启动过程均无须人工干预。

17.6.2 案例一：基于Symantec公司的应用容灾产品VCS①

图17-42所示为两台DB2数据库服务器，下面要将其配置为一个HA双机热备系统，主机硬件或者应用程序故障之后，由VCS自动检测故障，并在备份机上重新启动各种环境以及应用程序。

图17-42 两台服务器的双机系统

主服务器名称为dbsvr1，IP地址为192.168.0.1；备份服务器名称为dbsvr2，IP地址为192.168.0.2。

两台计算机操作系统都是Solaris 9，利用Symantec的Storage foundation（包含了VxVM和VxFS）作为卷和文件系统管理工具。

在两个系统中分别安装了DB2数据库程序，而数据库文件存放在共享磁盘阵列上面。共享卷由VxVM对底层磁盘进行虚拟化而生成。VxVM先将操作系统底层磁盘（盘阵上的LUN）组成磁盘组，然后在这个组中再划分卷，这就和RAID卡的做法类似，只不过VxVM作用在主机操作系统层，而RAID卡作用在硬件层。

本例所生成的共享磁盘组命名为DBDG，只包含一个物理磁盘，设备名为c0t0d0，然后划分一个卷，卷名DBVolume。用VxFS格式对这个卷进行格式化，再将格式化好的卷挂载到虚拟目录中。本例将其挂载于/home/db2下面。

对卷的划分和格式化仅需在一台计算机上配置即可，配置完后，只需要将这个卷进行导出操作，另一台计算机就可以导入并识别出这个卷的格式，再直接挂载到虚拟目录。

为了对客户端透明，我们用一个虚拟主机名和虚拟IP作为访问DB2数据库服务的地址。虚拟主机名为dbsrv，虚拟IP为192.168.0.3。

虚拟IP不是一个神秘的东西，我们知道一块以太网卡可以有多个IP与之对应，如果把192.168.0.3这个IP绑定到dbsrv1主机的网卡上，那么dbsrv1主机就同时拥有两个IP：192.168.0.1和192.168.0.3。这样，客户端用ARP协议请求192.168.0.3这个IP地址对应的MAC地址时，dbsrv1这台主机便会应答，客户端知道192.168.0.3这个地址的MAC地址（dbsrv1主机网卡的MAC地址），就可以建立与dbsrv1主机的通信。

一旦dbsrv1主机发生故障，那么dbsrv2主机上的VCS软件就会将192.168.0.3这个IP地址设置到dbsrv2主机的网卡上，并发出Free ARP广播，将新的IP与MAC地址的对应关系通告到网络上的其他终端。客户机再次连接的时候，就会建立和dbsrv2主机的通信，而客户端对这个IP的拥有者是dbsrv1还是dbsrv2丝毫没有察觉，也没有必要察觉到。这个切换IP的动作，也是VCS将虚拟IP作为一个资源来切换的过程。

提示：Storage Foundation的安装过程这里就不做描述了，本例假设在两台计算机上都已经成功地安装Storage Foundation组件

了。

相关配置配置过程

(1) 在dbsrv1主机上创建供DB2数据文件使用的共享存储及文件系统。

```
# vxdg init DBDG c0t0d0 \\创建磁盘组DBDG，使用c0t0d0这个硬盘
# vxassist -g DBDG make DBVolume 5g \\在磁盘组上创建5GB大小的卷DB
# mkfs -F vxfs -o largefiles /dev/vx/rdisk/DBDG/DBVolume \\将卷
格式化为VxFS文件系统
# mkdir /home/db2 \\创建挂载点，将用于DBVolume卷的挂载
# mount -F vxfs /dev/vx/dsk/DBDG/DBVolume /home/db2 \\将格式化
DBVolume挂载于/home/db2下，这样就可以通过CD /home/db2进入这个目录从
这个卷的内容进行访问了
```

(2) 使两个系统可以通过RSH方式互相访问，在dbsrv1上面做如下操作。

```
# echo "dbsrv2 192.168.0.2" >> /etc/hosts \\将对方加入自己的主机
# echo "dbsrv2 db2inst1" >> $HOME/.rhosts \\使得对方主机可以通过
以db2inst1的身份登录本机。Db2inst1是DB2数据库所必需的用户
# echo "dbsrv 192.168.0.3" >> /etc/hosts \\将虚拟主机加入自己的:
```

(3) 在dbsrv2上面做类型的操作，将dbsrv2改为dbsrv1，IP也做相应的改变，虚拟主机IP和主机名不变。

(4) 在两台计算机上分别执行下列命令，创建相同的用户组。

```
# groupadd -g 999 db2iadm1 \\创建DB2实例管理组;
```

```
# groupadd -g 998 db2fadm1  \\创建DB2 fencing管理组;
# groupadd -g 997 db2asgrp  \\创建DB2数据库管理组;
# useradd -g db2iadm1 -u 1005 -d /home/db2 -m db2inst1
\\创建DB2实例管理用户
# useradd -g db2fadm1 -u 1006 -d /home/db2fenc1 -m db2fenc1
\\创建DB2 fencing 管理用户
# useradd -g db2asgrp -u 1007 -d /home/db2as -m db2as
\\创建DB2数据库管理员账户
```

注意： 上述用户组或者用户的ID可以是尚未被使用的任意数字，但一定要保证两台计算机上面的用户ID是一致的，否则数据库切换的操作会失败；数据库实例管理员的账户目录要存放在共享盘上面，也就是/home/db2目录。

（5）在两台计算机上面分别安装DB2数据库程序。用install程序来安装DB2，然后手动创建实例和数据库。因为实例目录需要放到共享卷上，也就是/home/db2目录。

（6）安装完DB2程序后，分别在两台计算机安装DB2的许可证。

```
# /opt/IBM/db2/V8.1/adm/db2licm -a db2ese.lic
```

（7）在dbsrv1上面创建实例（存放到共享盘）。

```
# cd /usr/opt/db2_08_01/instance
# ./db2icrt -u db2fenc1 db2inst1  \\创建一个名为db2inst1的实例，I
将实例目录存放到同名的用户名目录下，也就是dbinst1用户的主目录：home/DB:
```

（8） 修改DB2节点文件/home/db2/sqlllib/db2nodes.cfg，将原来的

db2srv1主机名修改为dbsrv这个虚拟主机名。

```
0 dbsrv 0
```

(9) 创建数据库testdb。

```
# su - db2inst1  \\切换数据库实例管理用户;  
# db2start      \\启动数据库;  
# db2 create database testdb  \\创建新的数据库tdstdb, 由于当前用户  
db2inst1, 所以testdb数据库被创建在/home/db2目录下, 也就是共享卷上;  
# db2 terminate  断开与DB2服务后端处理进程的连接;  
# db2stop       \\停止数据库;
```

(10) 将共享盘从dbsrv1卸载下来 (在dbsrv1执行)。

```
# umount /home/db2  \\卸载文件系统;  
# vxvol -g DBDG stopall  \\将DBDG的所有卷停止活动;  
# vxdg deport DB2DB  \\将磁盘组DBDG导出, 以便在其他计算机上导入并挂载
```

(11) 将共享盘挂载到dbsrv2 (在dbsrv2执行)。

```
# vxdg import DBDG  \\将磁盘组DBDG导入;  
# vxvol -g DBDG startall  \\将DBDG的所有卷启动;  
# mount -F vxfs /dev/vx/dsk/DBDG/DBVolume /home/DB2  \\挂载文件
```

(12) 在dbsrv2启动原来在dbsrv1创建的数据库testdb。

```
# su - db2inst1  
# db2start  
# db2 connect to testdb
```

如果能够连接成功，则数据库双机配置成功。如果数据库服务在某系统上发生故障后，会被VCS切换到另外一台计算机并运行。下面配置自动故障检测并切换的功能。

(13) 复制DB2代理配置文件到VCS的配置目录。

```
# cp /etc/VRTSvcs/conf/Db2udbTypes.cf /etc/VRTSvcs/conf/conf
b2udbTypes.cf
```

(14) 打开VCS图形工具。

```
# /opt/VRTSvcs/bin/hagui  \\将运行VCS图形化配置工具
```

(15) 创建服务资源组（service group），并命名为db2grp。

(16) 依次单击“文件”→“导入”→“确定”按钮，导入DB2代理配置文件。

(17) 在db2grp中创建六个资源。

- 磁盘组：即DBDG。
- 卷：DBVolume。
- 挂载点：/home/db2。
- 网卡：客户端所连接的网卡（例如bge0）。
- IP地址：选择192.168.0.3这个虚拟IP地址。
- DB2 agent：这个资源会监控DB2程序在群集中的运行情况。

(18) 为这六个资源创建依赖关系（右击资源，选择link）。

IP依赖NIC网卡的工作正常； 卷的存在依赖于磁盘组的状态；文件

系统依赖卷；DB2代理的状态要同时依赖于IP地址的存在和文件系统的存在。

(19) 右击db2grp服务组，选择online，让db2在dbsrv1上线。

(20) 右击db2服务组，选择switch to，让db2切换到dbsrv2。

(21) 如果切换正常，则VCS配置成功。

17.6.3 案例二：基于Symantec公司的应用容灾产品VCS②

图17-43是两个站点容灾系统最基本的结构图。主站点是基于三个节点Cluster的多个应用，容灾站点同样也配置成一个三节点的Cluster系统，它们配了同等容量的存储，并具有数据容错功能。

图17-43 两个站点的互备群集

主站点运行的是Web服务、DB2服务、SAP企业ERP服务以及邮件转发服务的关键业务，完全置于容灾系统控制之下，可以看到：

- 主站点服务器A上安装了Web服务程序和SAP应用程序，而且二者皆在运行状态。
- 服务器B上安装了邮件转发处理程序，以及SAP应用程序和DB2数据库程序，但是服务器B上的SAP和DB2程序平时都处于停止状态，只有邮件转发程序在运行。
- 服务器C上安装了DB2数据库程序，并处于运行状态，另外还安装有Web服务程序和邮件转发程序，但是平时处于停止状态。

备份站点的业务是FTP服务、SMB文件共享的一般业务，不做容灾。备份站点的服务器A上运行的是FTP服务程序，同时安装有SAP应

用程序，但是SAP应用程序处于停止状态；服务器B上运行着SMB文件共享服务，同时安装有DB2数据库服务程序，但是DB2数据库服务程序平时处于停止状态；服务器C上安装有Web服务程序和邮件转发程序，并且都处于停止状态。

图17-43中最长的箭头指示了两个站点数据同步的路径，即从主站点盘阵（或者主站点服务器的内存），经过前端以太网交换机，传送到网关设备，然后经过广域网到达备份站点的网关设备，再通过前端以太网交换机传送到备份站点服务器内存，最后从内存写入后端磁盘阵列。

主站点服务器之间的箭头，表示一旦某个服务器，或者服务器上的某个应用发生故障之后，资源组的切换走向。

从图17-43中可以看到服务器B和服务器C形成了一个互备的系统，即服务器B是邮件转发程序的主节点，是DB2服务的备用节点；而服务器C是DB2服务程序的主节点，是邮件转发程序的备用节点。

提示： 因为主站点的三台服务器之间形成了比较复杂的互备关系，所以三台服务器必须能识别到其他两台服务器上挂载的卷；但是备用节点不应当挂载这些卷，仅当对方应用或者整个服务器故障的时候，才能在备份节点上挂载这些卷。

图中央的粗箭头，表示一旦主站点发生了大故障，诸如整个机房被损毁等，那么所有主站点的应用，全部切换到备份站点，并且备份站点的节点挂载备用磁盘阵列上的所有卷。Veritas的Storage Foundation组件应当安装到图上的所有服务器中，VVM模块用于管理所有存储卷，VVR模块用于同步所有卷的数据，VCS模块用于检测故障并且切换应用。整个HA系统的工作过程如下。

(1) 主站点应用的运行过程中，所修改的数据通过所有主站点服务器主机都安装的VVR软件实时地复制到备份站点。

(2) 假设某时刻主站点服务器上的SAP应用发生故障，比如相关服务无法启动，则VCS模块检测到这个故障之后，发现相关资源组有两个备用节点可切换：主站点的服务器B和备份站点的服务器A，所以它首先检测主站点的服务器B是否可用。如果可用，则发送一些信息通告服务器B上的VCS模块，准备切换SAP应用到服务器B，服务器B确认后，VCS在服务器A上首先卸载SAP程序所存储的对应卷，然后通告服务器B卸载成功，服务器B再挂载这些卷，并且接管SAP服务所利用的IP地址，之后启动SAP服务。客户端只需要重新连接一下便可。

(3) 某时刻，主站点机房供电系统故障，经过相关人员的检查，恢复供电大概需要5小时。而UPS系统在工作两小时之后因电量不足而停止，企业CIO果断决定，在UPS电量耗尽之前，将主站点所有系统手动停机以免因为突然断电对硬件和软件带来的损害。此时备用站点的VCS软件检测到了这个故障，立即在所有服务器上挂载已经经过数据同步的卷，然后启动所有备份应用系统。所有生产均恢复运行，客户端经过修改host文件或者修改所连接的IP地址，恢复了与服务器的连接，所有生产继续运行。

(4) 5小时之后，主站点机房供电恢复，UPS系统充电。企业CIO决定，在恢复供电1小时之后（确保主站点供电恢复正常，以避免不必要的动荡），切回所有应用到主站点。主站点所有系统开机，VCS软件会检测到当前的应用已经全部运行在备份站点，所以不会在主站点服务器上挂载卷并启动应用。与此同时备用站点的VVR软件重新建立了与主站点VVR软件的通信，并互相交互数据，备份站点的VVR检测到了主站点相关卷上的数据是落后的，因为备用站点在主站点故障期间，已经

生产运行了5小时，此间数据已经有所变化。VVR立即将变化的数据复制到主站点的相关卷。重新同步后，备用站点的VCS停止应用、卸载卷，主站点的VCS挂载卷，启动应用，所有状态恢复如初，客户端重新连接即可连接到主站点的服务器上。

本次故障造成的停机时间很短，没有对生产造成太大影响，同时也很好地考验了这个企业的异地容灾系统的功能。如果没有容灾系统，这个企业就要忍受长时间停机带来的损失。

17.7 虚拟容灾技术

传统的生产工具容灾方式下，对于服务器的容灾，通常使用配置相当的服务器作为备份机。但是如果有10台服务器需要做容灾，那么也要准备10台物理服务器作为备份机么？不见得。一种办法是只使用一台或者几台服务器，其上安装与主服务器相同的应用程序，主服务器有哪些应用，备服务器就安装哪些应用，安装在同一台或者几台物理机器上，当主服务器发生故障之后（存储故障、服务器自身硬件故障等），在备服务器挂载主服务器的LUN并且启动应用，有几台服务器故障，就在备服务器上启动对应的应用程序。这样虽然是一种做法，但是在服务器恢复的时候，需要重新安装操作系统重新部署应用程序，然后才能将备服务器上运行的应用程序切换到主服务器上，这是一个比较耗时的过程。

为了解决这个问题，虚拟容灾技术出现了。还是这10台服务器，如果利用VMware提供的P2V技术将整个服务器系统盘以及数据盘做成虚拟机格式，然后在一台或者几台备服务器上使用VMware根据生成的镜像来创建对应的虚拟机，并运行所有的虚拟机，在主服务器端使用一个Agent程序将对应应用系统对底层的IO数据同步到备服务器Guest OS上对应的存储设备中，这样，当主服务器故障时，备服务器可以直接启动应用，接管服务。主服务器恢复之后，利用VMware提供的V2P技术，将虚拟机对应的映像转换成物理机对应的数据并将其放置到对应的LUN中，这样，主服务器就可以直接启动操作系统并且启动应用程序了。

目前已经有一些厂商推出了虚拟容灾产品，比如爱数备份存储柜v3.5。

17.8 一体化先行军——爱数一体化备份存储柜

我们都知道电话/传真/复印/打印/扫描一体机，所有人对它的评价都是“好用、够用、实惠”，正如一句广告词所说的：“花一样的钱买五样！”如今在存储领域，也出现了这样一种一体机，即存储、备份、容灾三合一一体机。好么，这存储领域的老三样一下子被集成到一台设备中了，谁这么有本事？这就是爱数（EISOO）软件有限公司的“备份存储柜”v3.5产品。

说明： 2010年春作者有幸参加了爱数产品全国巡展青岛站，会议中与爱数的同行们进行了深刻学习探讨。感觉爱数是一家朝气蓬勃的公司，非常开放，这就注定了她会飞快发展。2009年下半年，爱数发布了一体化的备份设备，备份存储柜3.0。没想到时隔半年，再次看到了备份存储柜升级版本v3.5，能将备份、容灾和存储集成到一台柜子里，这需要投入很大的研发精力，爱数却做到了，而且是如此之快。

17.8.1 爱数备份存储柜3.5产品架构分析

我们先来看一下传统的备份架构。如图17-44所示，传统的备份系统中包含备份服务器、介质服务器（备份介质可以是磁盘阵列或者磁带库）、备份软件，也可以简称4S备份方案（Backup Server, Operating System, Backup Storage, Backup Software）。

图17-44 传统备份系统架构

我们可以把备份服务器和介质服务器的角色集成到磁盘阵列中，形成一种带有集成存储和备份功能的磁盘阵列，这就是爱数备份存储柜的

原型体。如图17-45所示，备份存储柜作为一台一体化设备被插入了系统中。它可以作为支持NAS、FTP、FC-SAN和IP-SAN访问协议的磁盘阵列设备而存在（相当于系统中多了一台磁盘阵列），同时还是一个备份服务器，用户服务器或者用户桌面电脑上的数据可以直接被备份到这台设备中存放，用户终端的桌面数据通过前端以太网备份，而用户服务器的数据则既可以实现通过前端以太网备份，也可以实现通过后端FC网络来备份（Lan-Free/Frontend-Free）。备份之后的数据还可以由这台设备再写到带库中离线保存，也就相当于D2D2T（Disk-to-Disk-to-Tape）。

图17-45 集成了备份服务器和介质服务器模块的阵列柜

一体化的备份设备就像防毒墙和防火墙设备一样，很快受到用户的欢迎，主要原因就是用一体化的方式改变了传统备份系统的模式，更加简单方便。之后，这个原型体开始进化，爱数将众多功能向其中融入。首先被融入的就是容灾技术。传统备份和容灾的一个本质区别就是，传统备份不是实时备份，RPO和RTO均太长；而容灾则是实时保护和灾难接管，保证业务连续性，RPO和RTO均显著缩小。

在容灾方面，爱数选择了与其他厂商不同的路线，巧妙地运用了虚拟化技术，即在备份存储柜上集成VMware Server版的虚拟机引擎，在存储柜上创建若干个虚拟机操作系统来作为环境中原先的生产物理机的后备服务器。物理机上安装一个数据实时复制代理，通过前端以太网来将数据实时同步到备份存储柜中运行的虚拟机磁盘中存放，当物理机发生故障之后，虚拟机立即接管物理机，继续服务，这就使得RTO变得非常短，理论上等于虚拟机接管物理机IP地址所耗费的时间。如图17-46所示为融入容灾功能之后的备份存储柜架构简图。

图17-46 融入虚拟机平台

对虚拟机的创建和管理可以直接使用爱数备份存储柜的配置界面，而无须使用VMware原来的配置工具，如图17-47所示。

图17-47 虚拟机管理界面

目前，爱数针对SQL Server和Oracle for Windows/Linux这两种数据库引入了文件数据块级（并不针对整个LUN，而是针对LUN中的文件，对文件进行增量块级同步，这样会避免很多不必要的数据传输，提高效率）的可感知应用层一致性的实时复制，相信随着时间的推移，其他可感知各种应用层一致性的实时复制均会实现。

然而，有了单纯的数据实时复制还不行，实时复制并不能保证数据的逻辑一致性，必须要引入Snapshot或者CDP技术，让备份后的数据可回溯。所以，爱数一步到位地在备份存储柜中引入了CDP技术。对相关文件的写IO对应的数据被从物理机同步到虚拟机中的虚拟磁盘（对应存储柜中的某个LUN）的时候，运行在虚拟机上的代理程序（用于接收物理机发送过来的数据以及插入文件系统驱动层来分流IO）会同时将其复制一份到CDP日志卷中保存（CDP日志卷由介质服务器模块所管理）。

提示： 但是爱数使用的这种CDP方式与本书其他章节中介绍的可实时挂载的CDP有所不同，这种CDP属于一种全局日志链（见本书对应章节）的模式，全局日志链极大地降低了CDP的处理难度，提高了效率。但是随之而来的也需要耗费额外的存储空间，而且日志链中的每一条操作不但需要记录当前被写入的数据，同时也需要记录对应的文件块被覆盖之前的内容，而这些记录中会有很多的冗余数据，基本上是冗余了一半，所以

日志链也将是很庞大的。（本书其他章节所介绍的非全局日志链CDP模型没有这种问题，请读者自行思考。）

这样处理之后，备份存储柜中不但保存了一份最新的与物理机源卷内容同步的可随时访问的虚拟机卷，同时也保存了所有针对这个卷的内容更改，一旦物理机发生灾难的时候，虚拟机直接就可以接管并直接读写虚拟机卷。如果一旦遇到数据不一致的情况，那么就需要利用CDP来回溯到之前某个时间点了。

具体的回溯方法是：系统将用户给定的时间点在CDP日志链中作匹配，找到对应时间点的位置，然后开始Undo（注意，是Undo）从日志链尾部到这个时间点之间的所有变更，将其全部回退，对应的回退动作反映在虚拟机卷中。回退完成之后，虚拟机挂载这个卷，应用启动，之后应用程序针对虚拟机卷中对应的文件所作的更改，也被追加记录到CDP日志卷对应的日志链尾部（称为虚拟机卷附加CDP日志链）并做好区分标记。这个附加CDP日志链记录了虚拟机接管之后所有针对对应文件的更改，在数据反向同步到物理机时需要用到。

同时，发生故障的物理机在故障修复之后，需要将数据反向同步到（通过前端以太网，由安装在物理机和虚拟机上的代理程序执行）物理机的LUN中，此时，系统根据虚拟机卷实际数据、源卷CDP日志链和虚拟机卷附加CDP日志链这三者，来把虚拟机卷当前的实际内容全部（通过读取虚拟机卷中相关的文件块以及Replay两个CDP日志链）复制到物理机对应的LUN中。（注意，爱数所实现的CDP为文件级CDP，日志链也只是针对对应的文件，复制的也都是对应数据库的文件数据，如果源LUN中有其他非相关文件，则不受保护。）

爱数技术： 在这里，爱数引入了又一项独有技术，即增量反

向同步。我们知道，物理机可能发生的故障多种多样，比如主板损坏等。这种故障物理机原来LUN中的数据是没有影响的（也有的时候意外宕机可能引起数据损毁），那么此时物理机卷中的数据与容灾端的数据大部分还是相同的，虽然容灾端接管后可能已经经过回溯过程，甚至已经被应用程序写入了新数据。在这种情况下，如果将容灾端的整个LUN或者文件都反向同步回物理机上，就是没有必要的了。爱数的做法是，对物理机尚存的源数据以及虚拟机所对应的最新数据影像分别计算Hash值指纹，通过比对指纹，找到指纹不同的文件块，然后只将这些块同步回物理机。

17.8.2 爱数备份存储柜v3.5独特技术

除了上面所述的一些关键技术之外，爱数还融入了一些其他独特的技术。我们知道CDP任意时间点恢复之后的数据并不能保证一致性，所以爱数在这里做了一些额外的工作来保证对应数据库的一致性。

爱数技术：前涉式一致性保证算法（Proactive Consistency）：是指使用某种API与数据库程序通信，让其处于一致状态，然后在对应的CDP日志链中标记此时的时间点，恢复时只要恢复到这个时间点，就一定是一致的。

启发式一致性保证算法（Heuristic Consistency）：是指通过监测文件系统一些特殊行为从而判断数据库的一致性状态，然后在CDP日志链中做相应标记，从而保证恢复时的一致性。

前涉式算法由于需要定时地对数据库发起指令让其处于一致状态，所以对数据库有影响；而启发式算法则一直处于旁路监测状态，不但对应用没有影响，而且其实现的一致性时间点粒度

也比前者要细。爱数使用启发式算法。

上文曾经分析过，全局CDP日志链导致数据占用大量的额外空间而其冗余度很高，鉴于此，爱数在介质服务器模块上使用了重复数据删除技术来消除。

爱数技术：爱数备份存储柜v3.5中的介质服务器模块所运行的Deduplication属于一种前处理式、全局指纹库的全局重复数据删除算法，这种算法对系统的要求非常高，由此可见爱数的技术实力。

爱数还提供了多点灾备方案。

爱数技术：提供二级级联容灾（P2L2R，Production site to Local DR site to Remote DR site），以及两地互容、一地多容、多地共容和链式环容的多点容灾拓扑。本地可以在执行完Dedup操作之后，再将CDP日志链数据同步到其他站点的存储柜中，这样就节约了大量网络带宽。

俗话说，三天不练只能瞪眼看。对于容灾，如果不进行演练，那么即便是部署的非常到位，灾难来临时也一样会手忙脚乱，天灾人祸一起来，业务恐怕就要中断很久了。而传统灾备架构中，要想来一次演练，那可谓是兴师动众，人心惶惶。但是有了虚拟容灾之后，事情就变很大了。爱数提供了两种容灾演练模式。

爱数技术：虚拟容灾的最大好处就是容灾端随时可用，并且可写（变化的数据有相应的CDP日志链记录，可以回退）。正因如此，用户可以随时找一台客户端直接连接容灾端的虚拟机，如果能连上而且做几笔业务发现没有问题，那么就证明容灾端至少在网络之下的所有层面都是运行正常的。整个过程对

物理生产机没有任何影响，而且所做的更改还可以回退。这就是爱数所提供的模拟演习，可以定期执行。

同时，实战演习爱数推荐一年进行一次。届时容灾端完全接管生产端，待回切时，系统将容灾端所做的变化同步回物理生产端。

值得一提的是，爱数的GUI做的非常好，所有对备份存储柜的配置均使用All-In-One-Web管理界面，通过浏览器即可登录管理，如图17-48所示。

图17-48 All-In-One-Web管理界面

说明：经过上述对爱数备份存储柜v3.5的架构和功能分析，你是否也会和作者有同感呢？还有很多细节的功能作者没有进一步研究。据作者所知，爱数目前有三大产品线，第一是备份软件；第二就是备份存储柜3.5；第三则是云备份平台。虽然“云”目前基本上还处于各家自行忽悠的状态，但是爱数能有如此的精力和实力来涉足云领域，可见爱数的技术水平以及鸿鹄壮志。可能由于孤陋寡闻，反正笔者看到爱数能够将如此多而强悍的功能做到一起时，佩服之余也感到欣慰，国产存储终将会辉煌！

17.8.3 国产存储的方向

我们目前尚不可能造出像西方EMC Symmetrix、HDS USP那样的大型高性能存储硬件产品，但是我们的智慧从来就没有输给过西方，古往今来，一向如此。《孙子兵法》、《本草纲目》、《伤寒杂病论》，哪一本都彰显了中国人对自然和对人文方面的智慧。但是由于长期封建统

治使得我们的智慧和科技受到了严重的禁锢。而如今新中国建国已经60年了，改革开放也已30年，科技文化等全面打破禁锢，飞速发展，我们应用智慧的时候也到了！

在计算机领域，最能体现智慧的地方就是软件。和西方人拼拳头，我们没有胜算；但是和他们拼脑袋，我们很有信心。存储硬件的性能再强，它也只是几个铁皮壳，输出为一堆LUN而已，而目前人们的需求越来越高，仅仅是提供高性能的LUN已经根本无法满足日益多样的需求，所以存储厂商近几年来正在向上层高附加值的软件方面投入大量精力；而对于硬件，也不再盲目追求专用高性能平台，而相继转向了开放的x86集群以降低成本，利用软件和集群的优势来弥补x86单节点性能不足的问题。这些都显示出软件方向已经成为全球存储行业的主导方向。

2008年左右存储行业两大巨头EMC和IBM都各自退出了自己的集群存储系统，这个事件将会是存储行业的里程碑，它宣布了从那一时刻起，软件从此上升到了主要地位。而这种方向的变化，就为我国的存储行业发展创造了不可错过的机会。软件方面，我国的研发人员水平不亚于印度人（国外存储厂商的研发人员有很大一部分都是印度人），而且最重要的一点，我们更加勤劳。所以，我们一定要抓住这种良机，迅速以软件解决方案占领存储市场。我们可以让我们的智慧凌驾于这些铁皮壳子之上，运筹帷幄，扬长避短。

爱数就是典型的以智取胜的例子，备份存储柜体现了中国人独特的智慧。通过软件和硬件的整合，爱数创新地打破了备份、容灾和存储的沟壑，创造了一台集存储、备份和容灾于一体的灾备设备，而且功能强大、实用又实惠，完全体现了国内用户的需求和接受能力，具有鲜明的中国特色。不仅如此，爱数备份存储柜的All-In-One-Web界面做的非常人性化，而且非常美观，这正是国内其他存储软件厂商所欠缺的地方，

这一点爱数可谓是用心良苦，中国人一般都爱面子，而爱数在界面上也给足了面子。

在不久前的全球气候大会上，中国向所有人承诺了节能减排的目标。而爱数的一体化产品也体现了绿色节能，与传统的解决方案一堆设备相比，一体化设备的节能与低成本显而易见。

爱数的创新是一种突破，也是一种趋势，顺应了第三代存储的发展方向，统一存储和应用存储的趋势正是包括DataDomain、NetApp等新兴存储厂商的发力点。

挑战国外的传统，既需要技术实力的突破，也需要应用和模式突破，包括备份设备和容灾设备在内的应用存储应该说是国内存储发展的一个重要方向。

17.9 Infortrend RR远程复制技术

Infortrend的RR（Remote Replication）技术支持双向及多站点复制，支持同步复制、异步复制，提供三种模式：

- Volume Copy模式可以让用户一次性将某个逻辑卷复制到远程使用，不提供后续的周期性数据同步。
- Async Volume Mirror采用异步快照滚动同步方式，周期性地将本地数据变化同步到远程。
- Sync Volume Mirror则是标准的同步数据复制。

如图17-49所示为Infortrend Replication Manager配置界面。可以看到常用的动作入口，比如创建/编辑/删除复制关系、网络状况测试工具、复制暂停/继续、同步/异步开始、断开复制关系、映射复制的逻辑卷、目标逻辑卷自动映射、切换复制关系。

图17-49 Infortrend Replication Manager界面

提一下比较有特色的两个地方。一个是提供网络状况测试工具，可以让用户一目了然的了解当前复制网络通路的带宽、时延、接收和发送的数据量等信息，以便灵活控制数据复制参数。如图17-50所示。

图17-50 Replication Manager网络状况测试工具

另外，Infortrend RR还提供了另外一种使用越来越多的应用场景的支持，也就是用同一台主机连接两台互为镜像的存储系统，要求当一台存储系统宕机之后，另一台存储设备无缝接管，拓扑如图17-51所示。这不但要求底层存储系统之间要维持数据复制关系，而且还要求多路径

软件充分配合。普通多路径软件不认为两台不同的阵列上报的逻辑卷是同一个，所以发生故障之后并不会自动切换。Infortrend提供Target Volume Auto Mapping功能，启用之后，一旦目标存储发现源存储系统不可用，会自动将复制的目标卷映射给预先配置好的主机，再加上多路径软件的配合，上层应用可以在很短的停滞时间内继续IO访问。

图17-51 目标卷自动映射

17.10 飞康RecoverTrac容灾管理系统

我们来做一回产品经理，如果让你设计一个容灾管理系统，你会怎么设计？先不考虑可行性，就按照你的理想来勾画。至少我是这么个思路：首先这个系统的主界面应该是一个可视化的、用图形化展示当前多个站点的各种状态以及数据复制链路拓扑关系，以及哪条链路上正有数据在被复制、时延多少、实时带宽多少。点击每个站点进入该站点内部视图，会显示出该站点内所有系统，包括主机/应用系统、存储、网络的运行状态。当发生灾难的时候，比如某主机宕机，而且一时半会无法上线了，业务又非常急，那么可以考虑在本地站点启用一台虚拟机来暂时代替这台主机，但是这要求之前主机的所有数据都必须存储在外部存储系统中存储，虚拟机启用之后，可以挂起这些数据卷，继续运行应用系统；如果是某站点整体宕机或者灾难，连容灾管理系统都无法登陆了，那么此时必须在容灾站点的容灾管理系统中强行在容灾站点启动对应的主机和业务系统，但是要求源站点的数据必须在容灾站点有一份，不一定也不指望是最新的。

一般产品经理也就像上面这样处理了，但是更加专业的产品经理需要将这些需求一层层地细化，比如怎么配置谁和谁是容灾关系，也就是A主机宕了，要起哪个备用主机/VM？依赖关系和启动顺序如何？演练如何实现？物理到虚拟、虚拟到物理、物理到物理、虚拟到虚拟，底层数据格式如何转换？如此复杂还是先洗洗睡吧！让我们来看看飞康的RecoverTrac容灾管理软件是如何解决这些问题以及如何设计展现的。

如图17-52上图所示为RecoverTrac部署示意图，RecoverTrac能够将NSS/CDP系统所提供的CDP、远程复制功能、主机端Snapshot Agent功能进行封装和自动化处理，然后再与各种应用进行适配，最终展现给用

户一个易配置、易管理的自动化容灾系统。支持V2V、P2V、P2P和V2P容灾。

图17-52 RecoverTrac部署示意图

如图17-53所示为RecoverTrac主界面，其支持物理机、虚拟机环境容灾，并支持原先由物理机运行的系统，容灾切换到虚拟机运行，反之亦然，当然虚拟机到虚拟机容灾、物理机到物理机容灾也不在话下了。该界面采用最为传统的左右分栏模式设计，左栏将多种资源分类，包括Host Image资源、Cluster Image资源、物理机、Hypervisor宿主机、VMware vCenter管理机、微软Hyper-V宿主机、存储服务器、站点、事件等。容灾管理系统，管理的其实是主机系统和存储系统，其本质上只做一件事，那就是将主机系统在另一个地方启动，其上的应用程序环境毫无变化，包括存储、网络环境。

图17-53 RecoverTrac主界面视图

飞康在理念上有一点很特别之处，就是将Host的概念脱离了底层“机器”的概念，也就是说，Host不等于机器，Host=应用程序+运行环境，而运行环境=主机名+操作系统+网络配置参数+存储路径（盘符和空间），此时，这个Host其实可以运行在物理机器上，同样也可以运行在虚拟机上，后两者才是机器。所以，脱离开底层机器的Host，飞康称之为“Host Image”，而底层的物理硬件或者虚拟硬件，飞康称之为“机器”也就是图中的“Machine”。

配置容灾系统，首先要把所有资源识别出来。这是第一步。如图17-54所示，首先创建站点，包括本地和远程站点，可以创建多个站点，然后在本地站点中将存储服务器（飞康NSS产品）、vCenter管理

机、虚拟机宿主机都注册进来，注册时需要输入IP地址和认证用户名密码等信息，因为系统需要从这些主机中获取虚拟机信息。每台Host中还必须安装飞康SAN Client客户端，从而与RecoverTrac通信，以便监控和管理容灾过程。

图17-54 创建站点、注册虚拟机管理机和宿主机、注册存储服务器

如图17-55所示为向系统中添加所有需要保护的物理机或者虚拟机，这就是为何系统需要先把vCenter、ESX Serve、Hyper-V宿主机预先注册和认证好的原因，因为系统会向这些主机查询虚拟机列表，从而方便用户选择添加。

图17-55 添加需要容灾保护的物理机或者虚拟机

其次，RecoverTrac还支持整个集群为单元进行容灾切换。所以必须先创建好对应的集群资源对象也就是图中的Cluster Image。如图17-56～图17-58所示。首先创建对应的集群资源对象，然后向其中添加Host Image，然后关联该集群中主机所使用的存储系统，最后关联该集群对应的虚拟IP地址。系统必须掌握上面的信息，因为，在做容灾切换的时候，系统会首先将所配置的存储系统逻辑卷的复制关系断开并切换到容灾站点，然后在容灾站点的机器上挂载好这些逻辑卷，然后配置好所设置的IP地址，最后才会启动对应的机器。

图17-56 创建集群资源对象并添加Host Image

图17-57 关联该集群底层所使用的存储逻辑卷

图17-58 告诉系统该集群的虚拟IP地址

如图17-59所示，配置完后的集群资源对象的属性可以随时更改。

图17-59 集群资源对象属性

在主界面的“Machine”中，定义的则是真正的物理机器或者虚拟机。物理机的加电启动也是可以由程序控制的，早期对物理机加电启动必须管理员在机器跟前才能完成，而如今各种远程管理硬件和协议已经非常完善，比如IPMI协议+BMC芯片共同配合，可以通过远程以太网来发送信号控制物理机的开关机。对虚拟机的开关机就不用说了，更加方便。能对机器开关自如，这一点是完成容灾自动化的前提之一。

向系统中注册物理机和虚拟机的过程限于篇幅就不再贴图了，基本过程类似。值得一提的是，不仅可以在RecoverTrac中注册已经存在的虚拟机，还可以直接在界面中创建新的虚拟机，因为前文中已经将vCenter管理机注册到系统中并认证了，所以RecoverTrac可以利用接口直接创建新的虚拟机。

创建好之后的物理机或者虚拟机对象的各种属性可以随时更改，如图17-60、图17-61所示。其中，物理机的Power Control属性页面中可以选择各种不同的远程管理协议；Service页面中可以选择如果该机器是Windows操作系统，其需要保护的服务列表，以便在切换到备用机器之后重新按照配置规则启动这些服务；SAN Client是指该机器所连接的飞康NSS设备需要对该主机做好映射关系，该主机作为该NSS设备一个SAN Client；Device则是指该主机所连接的逻辑卷，系统可以自动发现，也可以手动添加；Hardware Profile则是用来描述物理机关键IO设备，比如网卡、FC卡的驱动配置参数，以便系统参考导入备用机器。

图17-60 物理机和虚拟机属性

图17-61 电源管理设置和Windows服务设置

所有资源对象注册和配置完毕之后，需要创建容灾任务。如图17-62所示，首先需要创建一个容灾任务，选择需要进行容灾的Host Image，然后选择对应的容灾端机器，以及设定这些Host Image之间的启动顺序，图中的Delay Time便是来控制各个主机延迟多久启动，以便保证被依赖的应用先启动，依赖他人的应用后启动，当然，这些必须由人来告诉系统，系统自身不会知道这些逻辑的。

图17-62 创建容灾任务和选择容灾端机器

如图17-63所示为选择灾备端的磁盘，如果本地站点和灾备站点属于共享存储型的拓扑，那么图中就需要勾选“Same Disk Devices”，此时相当于仅仅可以容灾主机宕机；如果本地站点和容灾站点之间采用数据远程复制技术，那么切换到灾备端运行之前，存储系统就必须先把复制关系切换，同时逻辑卷也是主备关系，此时就需要选择使用“Replica Disk Devices”来作为灾备端所使用的逻辑卷。

图17-63 选择恢复用逻辑卷和调整启动参数

容灾任务创建完毕之后，便可以在灾难发生的时候，运行对应的任务，系统便会按照之前设置好的参数和恢复方式将宕机的Host Image重新在其他的机器上运行起来。

大致步骤如图17-64所示。当开始执行某个恢复任务的时候，

RecoverTrac会首先对当前数据卷做一份快照，然后会利用这份快照来作为灾备端主机的底层数据，为何不直接用原卷上的数据？因为此时灾难已经发生，原卷的数据最好别碰，碰坏了玉石俱焚，还是做一份快照来使用保险的多。快照生成之后，RecoverTrac会将这份快照挂起来，然后将其改变成灾备端主机硬件以及OS能够支持的格式，比如P2V、V2P、P2P，如果两台物理机硬件规格、HBA卡型号也不同，就可能需要做一些更改才可以成功挂起数据卷。然后，RecoverTrac会将处理好的快照映射给灾备端主机，但是先不启动灾备端主机。下一步，如果是计划内切换，RecoverTrac会首先停掉源端业务主机，然后启动灾备端主机，主机启动之后，数据卷自然挂起，业务自然启动；如果是灾难真实发生，那么系统会直接启动灾备端主机。

图17-64 恢复流程

说明： 虽说飞康RecoverTrac的配置界面并没有达到本人理想中的容灾管理系统的用户体验效果，不过其对容灾的封装抽象，比如，将Host作为一个资源，其底层可以对应物理机也可以是虚拟机，并可以任意P2V/V2P/V2V/P2P，其概念是非常独特的，对容灾的管理、监控和配置等方面可以说非常强大，有更多的细节可以去挖掘和使用。在此我也希望飞康研发团队可以将好东西宣传出去，界面上的改进在如今移动终端时代并非难事，欲行千里，好马须配好鞍。

国内也有厂商开发的类似容灾管理系统，但是无论是理念、技术实现、细节、体验等，都达不到飞康RecoverTrac的水平，而且有很浓重的参考痕迹，不再多表，究其原因，还是缺乏十年磨一剑登峰造极的执着，不重视技术，急功近利。振兴民族科技之路，谈何容易！

17.11 带宽、延迟及其影响

100Mb/s，这个速率意味着什么呢？有人说，每秒可以传输10MB的数据（8/10b编码下）。通常情况下，这种说法是对的。但是如果发送方与接收方之间的距离变得很远，比如数百公里甚至一千公里，那么这种说法，你会发现根本不成立。我们现在就来分析一下。

大家知道，光或者电信号的传输是有固定速度的，即近似每秒30万公里。（光在真空或者空气中传播可以达到30万公里每秒，但是在光缆中传播的实际速度为20万公里每秒，而电信号在电缆上的传播近似为21万公里每秒）如果两点之间距离为1000公里，那么信号传一个来回（传到对端，然后对端给以ACK应答）所耗费的时间就是 $1000 \div 300000 \times 2 \approx 6.6\text{ms}$ 。什么概念呢？也就是你想把1b的数据传输到一千公里之外的地方，那么至少你要耗费6.6ms。那么传输10b、100b、1Kb、100Mb，需要多长时间呢？首先想到的是，至少比传1b要慢。到底需要多长时间？来看这个公式：传输来回时间 = （数据量 ÷ 链路速率 × 2） + （传输距离 ÷ 光速 × 2）。数据在传输的时候，首先会被通过编码电路将数据串行化编码然后放到电路或者光路上传输，这个编码速率，就是链路带宽，100Mb/s的带宽与1000Mb/s的带宽，区别就在于后者在单位时间内可以编码相当于前者10倍量的数据。但是不管链路带宽有多少，数据被编码之后，数据在电路上的传输所耗费的时间对各种速率的链路来讲都是一样的，因为传输的时候已经与链路编码速率（带宽）无关了，传输到对方之后，对方还需要解码（所以编码所耗费的时间也要乘以2），同样也是取决于链路带宽。

所以，当两点之间传输距离很近的时候，比如1千米，那么传输时延 $\approx 0.0066\text{ms}$ ，基本上可以忽略了。所以那个公式变为：传输时间 =

（数据量÷链路速率）。所以说，链路速率越大，只代表其编码速度越快，而不代表传输速度越快，传输速度是固定的，都是光速。再打个比方，有一辆长途车，50个人排队上车，排队上车需要120s，汽车行驶需要60000s，50个人排队下车需要120s。50个人被排队送上车，就好比数据被串行编码放到电路上传输，汽车行驶相当于电路信号从一端传递到另一端，50个人排队下车，就好比对端的解码过程了，然而到此还没有结束，当汽车抵达目的地之后，司机必须在返回出发点进行报信，这就好比TCP协议在收到数据之后发送给源端的ACK应答一样。司机可以空着车跑回去报信（单独发送ACK应答包），也可以在目的端捎带着一些回程客人返回去报信（TCP可以在反向流量中夹带ACK应答信息以提高效率）。但是在容灾系统中，数据总是从源端流向目的端的，或者在灾难回切的时候从目的端流向源端的，总之只有一个方向有实体数据流动，那么此时回程ACK都是独立的ACK应答包（独立ACK包很小所以其编解码所耗费的时间也忽略掉即可）。

另外，一辆汽车能承载的人数是有限的，也就是说，得一趟一趟地拉，这就好比TCP每次所发送的最大数据长度，也就是TCP的滑动窗口长度，TCP得分批把用户数据传送出去，每次的发送量必须小于TCP滑动窗口的长度，每次传输之后均需要对方发送一个ACK（这里不考虑ACK合并等特殊情况）。每批数据虽然到了底层可能被切分，比如TCP的MSS（Max Segment Size）切片，一般等于底层链路的MTU，底层链路再用MTU的值来切片，但是这些底层的切片在被传输到对端之后，并不需要对端底层协议的应答，只有对端的TCP在完整的收到TCP发送的一批数据之后，才会应答。

那么我们来算算在相隔1000千米的两点之间，每秒到底能够传送多少个来回： $1000\text{ms} \div 6.6\text{ms} = 151$ 个来回。如果按照TCP的典型滑动窗口

即16KB来计算的话（每次发送16KB数据然后就等待应答，不考虑延迟应答或者合并应答等特殊情况），那么每秒吞吐量仅为 $151 \times 16\text{KB} = 2416\text{KB}$ ，也就是2.4MB每秒。夸张么？

当然，上述算式是忽略了编解码所耗费的时间以及整个链路上各种中继、转发或者协议转换设备所带来的处理延迟（好比长途车途中遇到的各种收费站、立交桥等）。如果算上这两者，则吞吐量会更低。更加准确的实际数据传输吞吐量计算公式为 $V = \text{TCP Window Size} \div 2$ （ $\text{TCP Window Size} \div \text{链路带宽} + \text{距离} \div \text{光速} + \text{链路设备处理延迟}$ ）。总之，距离越远，实际传输吞吐量就越低，在实际应用中一定要有底。

当距离很短时，可以忽略距离带来的延迟，此时显然谁带宽高谁传的就快；而距离很长时，此时带宽再高也无济于事，因为大头都被距离给耗掉了。另外，即便是底层链路的带宽相同，距离也相同的情况下，使用不同的协议进行传输，所带来的延迟也是不同的。但是设想一下，不管链路跨越了多长的距离，如果这条链路上永远都有数据在传着，那么发送方与接收方就可以以链路带宽的原生速率来收发信息，只不过有时延，就像卫星电视那样，此时传输速率并不会打折，如果做到这一点，那么对于一个容灾系统来讲是非常好的事情，充其量只会丢失几毫秒之内的数据。但是，事实却并非如此。超远距离传输，怕的就是数据流的卡壳，卡一次两次不要紧，频繁卡壳，那就根本无法利用起链路带宽了。这就好比磁盘寻道操作一样，本来磁头能以很高的速度读写盘片上的数据，但是没办法，必须换道，这一换道，外部速率骤降。碰巧的是，15K转每秒的SAS盘其平均寻道时间为5.5ms，而一千公里距离的传输时延为6.6ms，这两个值倒是接近而且还挺有意思。

传输协议无法避免“卡壳”，因为总要传一段歇一段来等待对方吱个声，看看收到没有。比如TCP，这样就平白无故的浪费了底层链路时

隙；再加上长距离下的高传输延迟，一来一回更浪费了大量时间，所以会出现上文中的即便是千兆链路下，1000公里的距离每秒也只能传输2.4MB的理论值，实际值将会更低了。

另外，如果在长距离下使用诸如iSCSI等协议的话，那将更是一笔惊人的浪费。大家知道SCSI层本身就有传输保障机制，人家自己有ACK那一套，而底层TCP再来这一套显然就显得多此一举了。按理说有了SCSI层的传输保障机制，其下层协议栈就应该是个无状态的类似链路层协议了，应该直接将数据一股脑传过去，但是现实是它非得传一段，停一段，等待对方说个OK，然后再传再停，慢慢腾腾；不仅如此，再加上SCSI也要传传停停，那就是变本加厉。所以长距离上跑诸如FCP、iSCSI等这种SCSI协议与FC/TCP/IP协议的合体协议，将会是个梦魇。

降低不必要的ACK数量，增加滑动窗口，这些都是广域网加速的技术，对传输速率会有一定程度的提高。但是最终解决办法，还是要尽量缩短两地距离，或者开发专用优化的协议了。

说到私有协议，这里就展开讲一下。上述所有场景，均建立在两点之间只有单TCP连接，即单流的场景下，此时的链路带宽当然无法被充分利用，而且也提过，如果底层链路一刻也不闲着，那么其有效带宽就可以更高的被利用，怎么办呢？显然，通过提高并发连接的数量，就可以充分利用起底层链路的时隙。

提示： 关于这个思想，在磁盘阵列控制器如何充分利用起后端FCAL环路的带宽方面也是类似的，大家可以阅读附录1中的第5问。

大家知道iSCSI里有个Multi Connection Per Session的概念，使用

Microsoft的软iSCSI Initiator的话，里面就可以进行设置，让Initiator端可以同时与iSCSI Target端建立多条并发的TCP连接，从而提高远距离传输时的效率，当然这个特性需要iSCSI Target端的支持配合。但是对于FCP来讲，就没有这种特殊考虑的并发连接设计了。经过考量设计的可并发连接的私有协议可以极大提高远程数据传输的效率。比如，在优质链路条件下，可以降低TCP连接数并且增加滑动窗口，而随着链路质量的降低，逐渐增加TCP并发连接数，同时降低滑动窗口的大小。

既然说到了多流并发，那么索性就再展开一些。对于一个异步模式的数据容灾复制系统，最起码要保证的是灾备端数据的一致性，而数据一致性又有多个层面，最底层的一致性就是所谓“时序一致性”，灾备端起码要保证每个IO都按照其在源端被执行的顺序刷入灾备端数据集中。如果使用单流TCP/IP则可以保证时序，但是传输效率很低；但是在多流并发的情况下，因为原本流与流之间是无关联的，可能在源端先执行的IO被传送到对端之后却被后执行了，此时就需要引入更复杂的逻辑来保证同步过去的的数据被按照顺序执行。这里又有两种办法可以考虑，一种是保证RPO，在多个流之间维护强一致性，将多个流强制关联以保证收发顺序，此时灾备端可以立即将收到的IO数据刷入底层数据集；第二种则是牺牲RPO，主备站点之间采用端到端的一致性组技术，在数据批与数据批之间保证时序性，而不是每个IO之间。此时灾备端不能在收到数据后立即刷入，比如等待一批数据全部收到之后才可以刷入。这么做虽然可能导致丢失一批数据而不是几个IO，但是可以方便的保证数据一致性。

第18章 鬼斧神工——数据前处理与后处理

■ 数据前处理

■ 数据后处理

■ 阴阳

存储相当于主机之下的主机，主机要做什么，存储都要做，但是存储做的，主机不一定做。存储上运行着各种各样的应用程序，这些程序专门管理和处理数据，作者发明了一个词，称其为“Data Cooker”，个人感觉非常合适用来描述这些程序，比如Snapshot、Deduplication、Mirror、Clone、Tiering、Migrating等。这些DataCooker可以在原本的裸数据之上实现更高级的功能，实现更多的附加值。

存储除了比主机多运行了一些更为强大的复杂的Data Cooker之外，其他与主机无异。所以，只要有了所有Data Cooker或者创造更强大的Data Cooker，那么就有了创造一台强大存储的能力。

相对于存储硬件来讲，软件更为重要，有些存储甚至连一些普通x86服务器的硬件指标都达不到，但就是因为它有强大的Data Cooker，可以让它称霸一方。如果有了更强大的硬件平台，可以将代码移植到新平台之上。不过看似目前不开放的平台正在萎缩，大家都在开放平台里给自己铺路，比如EMC的V-MAX，高端存储也用普通x86。

18.1 数据存储和数据管理

存储系统有两大部分内容：数据存储、数据管理。数据存储部分包括：存储控制器硬件、磁盘、适配器、网络传输通道、RAID管理、LUN管理等，这个部分的主要功能就是提供基本的裸数据的存储服务，比如将数据存储到对应的LUN中或者从其中读出。数据管理部分包括：Tier、Snapshot、Clone等数据处理功能模块，更多的数据管理功能模块如图18-1所示。

图18-1 数据存储和数据管理层次图

数据管理部分主要负责一些高层的数据处理，就像社会需求层次金字塔一样，最底层的需求是生存、温饱、有衣服穿，这一层就对应了存储系统对数据的基本存储功能，比如向一块磁盘中存储数据；再往上就是更高层的需求，比如不仅要温饱，还要吃好，味道足，而且还要物质丰富，要啥有啥。这就对应了存储系统对数据存储的进化，比如将多块硬盘做成RAID提高性能，并且划分LUN使空间分配更加灵活，硬件配置越来越强大；然而人的需求总是得寸进尺而且越来越疯狂并且无聊的，吃还要吃出花样来，吃出品位和精神来，吃完了还不行，还要追求精神层面的享受，比如，钱太多了，上街撒一把，或者装个乞丐乞讨。这就对应了存储系统对数据的更高级的处理，Mirro、Snapshot等，甚至一些华而不实的功能。

又可以把数据管理部分分为对数据的前处理和后处理两大类别。前处理指的是当数据还未写入磁盘之前就已经对数据进行了初期加工，或者对数据的存放空间做预先的准备，之后才写入磁盘存放，前处理的例子比如：Post Deduplication、Thin Provision等。后处理则指的是当数据

写入磁盘之后，功能模块将这些数据再读出进行处理，之后再写回磁盘。后处理的典型例子比如：**BackGround Deduplication**、**Data Migrating**等。有些功能模块既包含前处理过程，又包含后处理过程，比如**Snapshot**。**Snapshot**的生成是一个后处理过程，但是一旦**Snapshot**生成，那么每个受影响的IO也都会经过额外的前处理过程，比如**CoW**过程，之后才被写入硬盘；再比如**Mirror**，**Mirror**可以前台同步执行，也可以后台异步执行，前者则就属于前处理，而后者就是后处理了。

18.2 存储系统之虚实阴阳论

太始之初为太极，即先天地而生者，即道，即混沌无极。无极而至则分化为两极，一为阴，一为阳，阴者为浊为实质，阳者为清为气为能量。二者本为和合，但合久必分，阴阳分化导致对立，阴阳二者通过不断叠加和合，量的积累产生质的变化，而衍生了万物万象，所以万物万象皆表现为阴阳实虚表里上下大小等两极化。然而分久必合，两极相合的过程便衍生了生老病死沧海桑田。两极分分合合，万物轮回生生不息。阴阳既对立又相合，阴灭则阳毁，阳毁则阴灭。

现在的社会是虚进实退，实实在在的东西越来越少，越来越没人去做，而虚的东西却越来越多的人趋之若鹜。虽然表面看来风光无限，但殊不知，虚是要靠实来支撑的，阳气是要靠阴实来运化生发的，不管任何系统，如果系统中的实质不够，那么它所能运化出的能量也就有限。如果利用某些歪门邪道强制运化，那么必定会元气大伤，阳气再也不能运化，此时整个系统就表现为一个没有知觉和功能的实质，这种实质其本质已亡，这就是所谓“阳灭则阴毁”。如果系统的实质已经亏空，则阳气也无法生化，此即为“阴损则阳损”。所以当整个社会的阳膨胀到一定程度之后，物极必反，能量将走向坍塌湮灭沉淀，然后再逐渐积累阴实，然后阳再次逐渐被生发扩大。

对于存储系统来讲，既然作为一个系统，那么它与世界中其他系统都有类似的本质，阴阳和合而生，同时阴阳对立，互引互斥，共同推动自身的发展过程。对于存储系统，实质为阴，比如控制器、磁盘、线缆、柜子等所有硬件；而软件为阳，比如操作系统以及各种Data Cooker。软件与硬件之间就是一种阴阳关系，脱离了软件，硬件就是一堆废铁；而脱离了硬件，软件就脱离了存在的根本而灰飞烟灭。如果硬

件不够强大，那么软件运行的速度也就不够快。同样，在软件层次之内，也有阴阳关系，底层软件为阴，比如设备驱动程序、RAID/LUN管理程序；而上层的软件为阳，比如Snapshot、Clone等，如果没有底层的阴实的支撑，上层的这些能量就无法被生化。

磁盘上只有数据，不同的存储系统，样子可能一样，磁盘也都可以一样，但是它们各自的修行境界是不同的，有些系统拥有强大的Data Cooker，而有些系统则徘徊于基本的数据存储层面。

万物只要法于阴阳，就可以保证运行顺畅，一旦与阴阳变化规律相悖，则就会产生各种问题。对于存储系统也遵循这个道理。比如某系统拥有强大的硬件，但是它却没有强大的数据管理软件模块，只能够提供基础的数据存储服务，那么这样的存储系统就很难满足目前的业务需求；如果某个存储系统拥有强大的数据管理功能，但是它底层的硬件却是捉襟见肘，性能很弱，那么也就无法高性能地发挥出这些高层功能。

阴位于阳之下，阴是实质，阳是能量。数据存储为阴，数据管理为阳。目前的存储市场更加注重修炼阳，而对于阴的修炼已经逐渐弱化，大部分厂商已经将底层硬件架构转为开放廉价的x86架构，利用集群化来支撑上层对底层性能的要求。

18.3 Data Cooker各论

由于Snapshot、Clone、Disaster Recovery、Virtualization、Mirror、CDP已经在本书其他章节介绍过了，所以下面只介绍Thin Provision、Deduplication、Tier、Space Reclaiming等。

18.3.1 Thin Provision/Over Allocation

Thin Provision目前被广泛的翻译为“瘦供”或者“自动精简配置”，对这两种翻译笔者个人是不赞同的。一是这两种翻译根本无法反映这种技术的本质，让人根本看不懂；二是这两种翻译容易让人理解错误，会误导别人。前者直接按照字面翻译，完全没有任何意义，后者则故弄玄虚，弄了一个让人摸不着头脑的看似“专业”的词。笔者认为Thin Provision直接翻译为“超供”最能反映这种技术的本质，即Over Allocation。Over Allocation的概念最早是由STK公司于1992年提出的，后来DataCore公司根据Over Allocation的思想在2002年提出了Thin Provision的概念并将其应用于存储产品，随后3PAR也于同年在其存储产品中应用了Thin Provision。

所谓超供，还是举个例子来讲，比如某台存储系统只有10TB的物理存储空间，现在有10台主机客户端，每个客户端各自需要1TB的存储空间，所以这10T空间只能给这10个客户端来用，此时已经没有剩余空间了。但是在使用过程中却发现，这10个客户端中，不是每个都能很快就将自己的1TB空间用完的，这就造成了空间闲置。有没有办法把这些闲置的空间利用起来，但同时又不能让原来的10个客户端感知到呢？

从技术上来讲，实现这种善意的“欺骗”不成问题。假如10个客户端

每个实际使用了500GB的空间，这样的话，整个系统内还剩下5TB的空间没被实际占用，那么此时存储系统可以在后台悄悄地将这5TB的空间再分配给其他客户端，比如，再分别分配1TB的空间给5个客户端，现在所有15个客户端每个客户端所看到的额定空间都是1TB，但是实际物理空间却只有10TB而不是15TB。这就是“超供”的“超”字所体现的含义。

更进一步，上面的例子，即使剩余了5TB未占用空间，那么既然存储系统可以自欺欺人地将其分配给5个客户端每个1TB，那么为何不能更加厚颜无耻地分配给10个、20个客户端，每个1TB，或者每个10TB的空间呢？本来已经是在骗，骗1TB也是骗，10TB也一样是在骗。最后，干脆变成一个彻头彻尾的骗子得了，骗子的最高境界就是连自己都被自己骗了。比如，本来自己只有10TB物理容量，但是它却通告自己有10PB的容量，所有客户端的分配空间，将会在这10TB的整体物理空间中按需分配，也就是说，分配给你多少空间只是虚的，你实际用多少才是实的，才是占用实际空间的。这种做法是基本常识，比如网络硬盘、邮件服务商等，你申请一个2GB的网络硬盘或者邮箱，供应商才不会给你预留2GB的空间，它只是记录一下而已，当你真的塞入了2GB的数据后，你才真正地占用了供应商的存储空间。

存储系统实时监控物理空间使用情况，一旦所有用户整体空间消耗达到临界值，则需要马上扩大物理容量。然而，对于空间使用率的监控方面，如果存储系统为NAS系统，提供的是一个基于文件协议的卷共享，则存储系统本身就可以很容易地监控存储空间的真实耗费情况，因为NAS系统是自己来维护文件与物理空间对应关系的。但是如果存储系统提供一个基于Block协议访问的空间，比如FC或者iSCSI协议的LUN，则存储系统所监控到的这个LUN在物理磁盘上所占用的空间使用率很大

程度上是伪的，存储系统监测到的占用率永远大于其实际占用率，其原因是因为存储系统自身一般不能感知到这个LUN中的文件系统在实际文件所占用的空间，只有客户端主机才能看到，如果在使用这个LUN的客户端主机上曾经将数据塞满整个LUN但是随后又删除掉了，那么存储系统所看到的这个LUN的使用率永远都是100%，而实际却是0。但是从技术角度来讲，存储系统想要监控LUN之内的实际数据使用率也不是什么难题，只要能够感知其上的文件系统逻辑即可探查得到，并且已经实现，下一节将会描述。

存储系统对LUN实际占用空间的监测，可以选择简单模式、复杂模式和完美模式。

在简单模式下，系统将记录一个High Water Mark，即目标LUN曾经接收到的写IO所对应的LBA地址中最长（最远）的那个，利用这个HWM来判断目标LUN实际占用的空间。比如，一开始存储系统创建了一个大小为1TB的LUN，但是由于使用Thin Provision模式，这个LUN刚被创建好的时候，实际是不占用存储系统物理硬盘的空间的，系统只是将这个LUN做记录，但是不为其分配实际空间。当主机客户端挂载这个LUN之后进行写IO操作时，假设第一个IO就是向这个LUN的最后一个LBA地址中写入数据，那么存储系统此时就会判断这个LBA地址，并且查看当前已经分配的物理空间尽头的LBA地址，如果当前已分配空间尽头LBA地址小于当前写IO的LBA地址，那么存储系统就会再分配一段长度为（当前IO的LBA地址-当前已分配空间尽头LBA地址）的物理空间给这个LUN。在这个例子中，虽然主机只发送了一个写IO，但是存储系统却需要分配与LUN标称大小相同的物理空间，这显然是不合算的，但是也是最高效的。

复杂模式下，系统可以识别简单模式所不能识别到的信息，要识别

出当前LUN真正需要的物理空间，就需要记录更多的信息，比如LUN中哪些地址被写过，哪些尚未被写过，而记录这种信息的最高效做法就是利用Bitmap。Bitmap中每个比特可以表示一个Block，或者一个LBA，但是限于效率方面考虑，每个比特表示一个Block（或称Page，比如4KB、16KB大小）是比较划算的。利用这个Bitmap，不但可以随时监测对应的LUN中哪些Block实际被占用，而且还能统计出实际占用的比例以便在将要达到物理空间极限时发出通知。有了这个Bitmap，系统就可以针对每个写IO都来参考这个Bitmap，如果当前写IO的目标地址对应的比特为1，则表示已分配其物理空间，则这个写IO直接写向对应的物理空间；而如果对应的比特为0，则表示对应的Block尚未被分配，那么系统就会先分配物理空间给这个LUN，更新LUN的Metadata，然后将数据写入对应的Block。这样做虽然效率低，但是能够节省更多空间。然而，就像上文中所述的，如果主机客户端确实曾经对这个LUN中的每个地址都写过数据，那么此时存储系统就会分配与LUN标称空间相同大小的物理空间。但是此时这个LUN内的所有数据可能都是“尸体”，即这个LUN之上还有一层逻辑在做映射，也就是文件系统（或者其他程序自身管理的数据映射机制）。纵使文件系统将这个LUN中的所有数据都删除了，那么此时对于存储系统来说，它根本无法感知到文件逻辑，所以此时这些被删除的文件依然是占用物理空间的。解决这个问题的办法将在下一节描述，也就是所谓完美模式。

Thin Provision的底层技术实现方式其实并不复杂。大家都知道，不管是在Windows还是Linux操作系统下，NTFS或者EXT3文件系统，都支持Sparse File，文件系统为每个Sparse File都保存了一张Bitmap，而这个Bitmap的作用正如上文中所述（关于Sparse File的更多介绍请参考本书其他章节）。所以，如果某存储系统直接使用文件系统来作为LUN的管理平台，那么它完全可以把LUN设置为一个Sparse File。但是由于Sparse

File并不支持超供，比如物理空间只有100GB，那么就只能创建100GB大小的文件，Sparse的作用只是在向文件写入大量的0x00时不会向磁盘写入这些0，而且随后的读出过程也不会产生原本应该发生的实际磁盘IO，而只是在内存中生成这些0。所以，要实现超供，必须对Sparse File功能加以增减。

而一些存储系统并不使用文件系统来管理LUN分布，那么这些存储系统就比较难受了。举例来讲，我们先来看看如果使用横向条带化方式来分布LUN的情况下（最传统的方式）系统是怎么实现Thin Provision的。显然，第一个能够想到的方法就是以一個或者若干个连续条带为单位来进行动态扩充，比如某时刻用户下发了针对某地址的IO，系统便会在RAID组中某处分配一个或者几个连续的条带，随后每次IO都如此，并且维护一个逻辑地址到物理地址的映射表。一开始系统尽量保证物理地址与逻辑地址是一一对应的，就是说逻辑上连续的地址在RAID组的物理地址也被连续存放。但是随着RAID组中LUN数量的增多，由于是Thin Provision，其他LUN的空间可能随时会挤占某个LUN的物理空间缝隙，这样下去之后，整个RAID组中的多个LUN之间的条带可能就变成一种错乱排布的情况，这样直接导致了逻辑连续的条带在物理上却不连续，在大块连续IO情况下却表现为随机IO的效率低下。

Thin Provision杜绝了浪费而避免了额外开销，降低了成本。而且，实现Thin Provision不需要多少配置过程，用户只需要在创建LUN的时候加一个标记即可，其他都是系统后台自动完成的。

注意： 但是使用Thin Provision时必须严格注意的是：随时监控目标LUN的实际占用空间和系统剩余的物理空间，当达到阈值时，一定要扩充物理空间。否则一旦发生数据溢出，主机端报出的错误将是类似“某某扇区写入失败”的底层严重错误，而

不是类似“磁盘已满，清除垃圾后再处理”之类的上层错误，可能将会导致一系列更加严重的连锁反应，影响主机端的运行。

1. Thin Provision对性能的影响

Thin Provision会产生一定的性能影响。有人可能产生这个疑问：“本来500GB的需求，你只给50GB的空间，相当于50GB的空间承载了10倍的IO请求，性能会成问题。”这个结论是在某论坛某网友提出的，笔者第一次看到时真的有点觉得被噎了一下。Thin确实对性能有影响，但是这个结论就属于完全的谬论了。假设不使用Thin，创建一个500GB的LUN，就给他预留500GB的物理空间，那么此时，这个LUN所接受的IO目标地址如果都落在50GB的地址范围之内，那么按照上面的结论，岂不是依然是“使用50GB的空间承载10倍的IO”么？所以说这个结论着实有些荒谬。

那么Thin对性能的影响到底体现在哪些方面呢？主要是两个方面：耗费额外CPU周期、物理空间碎片。

开启Thin模式之后，针对LUN的每个IO都需要耗费额外的处理流程。比如，还是上面的例子，50GB物理空间，当这个LUN接收到一个超过50GB地址范围的IO时，比如是读IO，那么Thin引擎就会先查询IO的目标地址是否已分配了物理空间。本例中尚未分配，所以Thin引擎会向上层返回全0x00，因为目标地址尚未分配物理空间，证明它尚未被写过，那么对应地址上的内容当然就应该都是0x00了；如果是写IO，那么Thin引擎也需要判断目标地址是否已经被分配了空间，如果已经分配，则直接将这个写IO导向对应的物理空间地址，如果尚未分配，那么Thin引擎还需要在整个物理空间内查找剩余的空间，而且还需要尽量保持与已经分配的空间在物理上连续。这一系列的判断和处理，加上还需要同

时维护一些元数据之类，都是需要耗费额外的计算资源的。

由于Thin不会预留标准的物理空间，而是随用随分配，就像操作系统对内存的管理一样。那么就无可避免地会产生物理空间的碎片，针对一个LUN，传统模式下是连续分布在物理空间之上的，而Thin模式下，可能这一块，那一块，因为原本应该连续被分配的空间很有可能被其他的Thin LUN所占用，多个Thin LUN混乱地分布。这样就导致了IO性能问题，本来上层的连续地址IO，经过Thin引擎处理之后，可能却变成了随机度大增的IO类型。这一点对于那些物理空间分配粒度很小的Thin引擎更为明显。

上文曾经提过，Thin可以有不同的细节设计。比如一个50~500GB的Thin LUN，当接收到一个位于80GB偏移处目标地址的IO时，Thin可以智能地判断上层在接下来的时间里，很有可能需要对其附近的地址有更多的操作了，所以，Thin引擎此时可以以80GB这个逻辑目标地址为基准，在其左和右各分配比如16MB/64MB/128MB的物理空间以预先占位，新分配的空间，视剩余物理空间的比例而定，尽量保持连续。或者Thin引擎干脆可以以更大、更简洁的办法处理，把分配粒度设计成最大化。

比如刚才那个例子，可以直接将50~80GB之间的30GB的物理空间分配下去，只不过这样做的肥胖率会大增，但是却节省了计算资源，同时降低了IO性能损耗率。Thin引擎不可能完全做到最细的最小化粒度分配，即以单个IO的目标地址为粒度，还是刚才那个例子，如果IO长度为1，即只读取或者写入一个扇区的话，那么Thin引擎不会只对这一个扇区进行物理地址空间分配，这样做效率太低，需要维护的Bitmap等元数据粒度太大，尺寸也就过于庞大，查找起来非常耗费资源，而且底层产生的碎片更多。所以，流行的还是折中的做法，即在IO目标的左右以相

当长度的粒度将空间分配下去，这样就可以在一定程度上保证连续IO的性能，同时可以保证耗费较少的计算资源。

说道Thin对性能的影响，这里有一个极端的例子，请看表18-1。这是国外某知名存储厂商某终端阵列开启Thin功能之后的性能下降统计表，可以看到最差情况下性能竟然下降了70%，而且是随机IO的情况下。如果说Thin会让逻辑上连续的块底层变得随机，那么这理论上应该不会对随机IO产生这么大的影响，因为原来也是随机，现在依然是随机，随机度并没有加成。那么原因出在哪里呢？

表18-1 Thin对性能的影响

这还得从Thin的源头原理来追溯。要想实现Thin的核心模块，只要有三大元数据基本就足够了：一个位图、一张表、一棵树。位图是用来记录底层连续物理地址空间内哪些地址已经被分配，哪些没有被分配的，以便用来迅速分配空间；表则是用来记录各个LUN的逻辑地址与物理地址对应关系的，因为逻辑地址不是传统那种与物理地址一一对应的关系了，可能随时被分配到任何地址上，这张表需要按照逻辑地址排序从而便于后续的查询操作。树则是在内存中生成的用于迅速查表的结构了。说到这，我们先来算算这张表的大概容量吧，假设使用64b的地址长度，分配粒度为32KB，也就是每32KB的物理块用一个64b地址表示，那么这张表内每一项条目的大小就是128b。对于一个容量1TB的空间来讲，需要33554432个32KB的块组成，那么这张表的容量就是 $33554432 \times 128 \div 1024 \div 1024 \div 8 = 512\text{MB}$ 的空间，而针对一个10TB的空间就需要保存5GB的元数据。而一般情况下，支持Thin的产品一般都会将所有磁盘建立成一个全局存储池，也就是说，系统中的全部磁盘都需要加入这个池，那么加入100TB的空间，就需要对应50GB的元数据。而如果使用1MB的块粒度，那么大家自己算一下，元数据会骤降到

16MB/TB。如果将地址长度也适当降低，比如降低到48b，则也能节省一定容量的元数据。顺带提一下的是，这种元数据量相对于文件系统元数据量来讲，还算是少的。正因为元数据如此之大，所以不可能全部驻留内存，这样就牵扯到每次上层的用户IO都可能需要读写这些元数据，所以会导致性能严重下降，也就是被读写惩罚所拖慢了，即便是随机IO，其性能依然有很大下降。至于元数据所耗费的计算资源，并不是很大，查询效率还是很高的。再加上传统阵列厂商在类文件系统方面积累不够，IO优化还是不到位，所以会导致性能相对下降。

HDS公司在USP V产品中使用42MB为一个粒度，EMC在DMX上的粒度为768KB，IBM在其SVC产品上的粒度为32~256KB，“IBM在其DS8000和Storwize V7000中的Easy Tier的粒度为16MB~8GB，默认为256MB”。而3PAR公司的粒度是16KB；然而这并不是业界最小的，最小的粒度当属NetApp的WAFL，为4KB，因为它对待LUN就像一个普通文件。3PAR公司的Thin处理模块被嵌入在ASIC中以加快执行速度。

2. Thin Provision的脆弱性

Thin的第一个脆弱性就是其对性能的影响实在不可忽略，甚至有时不可容忍。某厂商的存储系统，开启了Thin功能之后，性能最大下降了70%，也就是性能最差时仅为非Thin LUN的30%。究其原因可以参考上文中的描述。这使得Thin很难被广泛地用起来，除非能够解决性能问题。而粒度与性能总是一对矛盾，为了保证性能而增加粒度，又会丧失Thin的意义，所以如何在性能与粒度之间取舍，是厂商应该考虑的问题。

Thin LUN需要时不时地进行碎片整理以保证性能。上文中描述了Thin导致的数据乱分布问题，这样的话，就需要考虑引入LUN碎片整理

机制。不要将LUN碎片整理与LUN文件系统碎片整理相混淆。后者是指使用这个LUN的主机端上的文件系统对LUN内的逻辑上的文件进行碎片整理，而前者是指将本身被乱分布到RAID组之上的LUN碎片进行整理，其实这两个层面的碎片整理本质上是一样的，可以把LUN看作是分布在RAID组之上的一个大文件。NetApp的WAFL就是这么做的，而且WAFL天然可以实现Thin LUN，现用现分配空间对于文件系统来讲是小儿科的事情，LUN就是WAFL下的一个文件。也正是因为如此，WAFL同样面临着碎片问题，为了保证性能，其提供了LUN Reallocation操作，也就是碎片整理操作。

Thin的第二个脆弱性便是其瘦身的效果很难维持，时不时地就会变胖。比如，主机端应用程序或者文件系统的一些行为。最典型的便是文件系统碎片整理过程，会首先读出零散的碎片，整合之后写入新位置，然后标记之前的位置为空闲。而“写入新位置”这五个字，是Thin LUN最不愿意看到的，因为这意味着系统要分配这些新位置，那么Thin LUN就逐渐变胖了。如果原本100GB的LUN，上面有50GB的碎片文件，而此时这个Thin LUN可能只实际分配了70GB的物理空间，那么当进行文件系统碎片整理之后，极端情况下，文件系统可能要读出这50GB的碎片，然后再写入50GB。假如碎片整理过程需要的新空间为30GB，那么系统一定就会在底层将原本的那瘦下去的30GB分配出来，此时这个LUN就是一个胖LUN了，就算文件系统碎片整理完毕之后，实际文件占用仍然为50GB，那么这个LUN也瘦不回去了，除非有回收机制（见下文）。

另外一个典型例子便是生成大量临时数据的应用程序。比如某应用程序运行时，某个条件下触发大量临时文件数据的生成，结束之后就删掉，那么此时对应底层的LUN来讲，也要分配对应的空间来存放这些临

时文件数据，虽然最后的状态是这些临时文件全被应用程序删掉了，但是底层已经胖了，回不去了。上一段曾经说过“Thin LUN需要碎片整理”，这一段又说“碎片整理会导致Thin LUN变胖”，乍一看非常矛盾，其实还是如上文所述，两层碎片整理各有各的功效，但是确实会产生对立面。

综上所述，Thin Provision真是一个脆弱的东西，如果只靠忽悠、掩耳盗铃来硬说自己瘦了，还是不靠谱啊！要真正不反弹的瘦，就要从应用层业务层入手，从源头上杜绝垃圾数据的产生。

3. Thin底层设计中的一些算法

累加模式的颗粒分配算法——比如T1时刻一个新创建的LUN1有针对其LBA1的写IO，那系统就写入Thin Pool的LBA1；T2时刻有针对新创建的LUN2的LBA100的写IO，那就写入Thin Pool的LBA2，依此类推。Thin Pool就像水池一样，不管接收到哪个LUN的追加写IO（之前尚未写过的地址），那么每个IO都会导致Thin池里的高水位线往上涨一个LBA号。

全随机模式的颗粒分配算法——累加模式的颗粒分配算法下，LUN的数据不能够被分配到系统内所有物理盘之上，不能充分利用起系统整体性能，所以可以使用全随机模式的颗粒分配算法，让分配范围弥散在整个Thin Pool的地址空间内而不是累加。

自适应分配力度算法——系统可以智能感知上层IO的行为，比如大块连续追加写入操作，那么此时就没有必要每次分配很小颗粒的空间，完全可以加大分配力度，比如一次分8MB、16MB，根据IO特点和持续时间等因素，不断提高分配力度；反之，如果检测到追加写IO负载

趋缓，那么就逐渐降低分配力度。

位移保持颗粒分配算法——分配存储空间颗粒的时候，尽量按照原来的相对地址分配。比如用户在一个空Pool里创建了一个Thin LUN，向LBA1写了数据，那么程序就向Pool里的LBA1写，后来又向LBA1000写了，程序也向1000写；后来又创建了一个Thin LUN，也向LBA1写，那么此时程序可以向Pool里的LBA2写，写在LBA1旁边，依此类推，LUN3的LBA1就是Pool的LBA3。但是一旦如果遇到Pool的LBA1和LBA2都被同一个LUN占据，那么程序就考虑寻找相对距离最近的LBA写入，总之尽量保持所分配的地址的相对偏移量与这个IO原本的目标地址相对偏移量一致。这样就可以保持逻辑上连续的颗粒在物理上也相对连续，对后续连续地址读写操作有好处。同时用户一般不会创建大量的LUN，所以颗粒地址争抢地址冲突的几率也不会很高，因此最好能保证颗粒块该放哪还是放哪，相对位移保持一致。

思考：累加分配模式下，由于系统会针对所有进入的首次写IO做连续地址写入操作，所以对外会表现出较高的性能，但是随着数据写入的推移，追加写越来越少，会逐渐演变为覆盖写，那么此时底层会变为完全随机地址IO，性能会降低，尤其是在上层连续地址IO时，底层总是表现为随机地址IO。

全随机模式的颗粒分配模式下，相对累加模式来讲整体性能有提升，但是对于初次/追加写没有任何加速作用。并且对于后续连续地址IO一样也会有所拖慢。

而位移保持分配模式下，由于地层的数据排布与原本的逻辑排布保持尽量的一致，所以由Thin的引入而导致的系统性能变化幅度会被控制到最低。而且对于后续连续地址IO是可以保证最大幅度的维持原生性能的。

综合来讲，应当使用使用位移保持颗粒分配算法。

颗粒格式化算法——上面的文字都是用LBA来指代分配颗粒的，实际上不会有哪个产品的颗粒度细到LBA扇区的。比如就拿768KB（EMC在其Symmetrix V-Max上使用的粒度）作为一个颗粒粒度，每次分配必须是32KB的整数倍。但是这里就会有一个问题，如果主机下发的IO只有4KB，那么底层也一样需要分配768KB，那么这时候问题就来了，这768KB中剩余的762KB都是僵尸数据，如果主机后续发起对这768KB地址范围内的读IO，此时Thin处理模块就不会返回全0x00了，因为已经分配空间了，必须要从磁盘读，这样的话就会把僵尸数据返回给主机，而不是原本应当返回的全0x00。为了解决这个问题，在颗粒分配之后，Thin模块必须做初始化操作，也就是将颗粒内没被当前写IO所覆盖的地址上的数据扇区写入全0x00。如果你真的这么认为，你就错了，底层根本不需要为Lun进行清零操作。一块用过的物理磁盘上，全都是僵尸数据，同样，一个新划分的Lun，其上也可以都是僵尸数据，底层没有义务清零。有人问如果读出来僵尸数据怎么办？答案就是没办法，如果一个程序不去写自己的数据到磁盘上这个区域，上来就读这个区域，那么这个程序不是测试软件就是数据恢复软件，正儿八经的软件没这么干的，都是先写入，再读，至于房间里面之前住了什么人发生了哪些事，人家不去操那个心。

强调：但是必须强调的一点是，一个Lun在被创建之后，其零扇区必须先被主动清零，因为零扇区太重要了，如果不清零，而发生了小概率事件（比如恰好底层磁盘的某零扇区内容刚好对应了该Lun的零扇区），那么此时就会出问题了，OS会直接按照零扇区里的分区表尝试挂载磁盘分区，会产生不可预知的后果。

后台重整机制——此外还可以提供一个后台重整功能，将乱序的颗粒重新整理成按照原来顺序存放以保证后续IO的性能。

在同一个地址池内同时支持Thick与Thin LUN——当在Thin地址池内创建一个Thick LUN时，系统就会直接在Bitmap中将对应的空间占住。之后针对这个Thick LUN的所有IO也都会Bypass掉映射表，其效果与普通LUN完全一致了。

18.3.2 LUN Space Reclaiming（Unprovision/Deprovision, Get Thin）

试想这样一种情况，比如某个存储系统只拥有1TB的物理存储空间，有多台主机客户端的LUN分布于其上，都占用了一定的空间，当前剩余物理空间只有500GB了。而此时，某主机客户端需要一个500GB的LUN空间，存储系统管理员只好将这最后的空间分配给了这台主机使用。这台主机其实只是将这个LUN设置为某程序运行时的tmp空间，其中放一些临时性的文件，不会超过500GB，此外，还存放一些其他用户层面的文件。当程序运行完毕之后，临时文件也会被删掉，但其后果就是，这些文件曾经占用的空间会永远地占用存储系统中物理硬盘的空间，也就是说这500GB的剩余空间，眼看着就白白浪费了。而且此时系统中剩余物理空间为0，存储系统无法对这500GB的空间进行Unprovision或者称Deprovision。Deprovision的过程又可以形象地被称为“Get Thin”过程。

虽然此时可以在存储系统端使用缩小LUN容量的动作来强制将这个LUN的大小降低到某个值，但是这样做风险很大。LUN缩小都是直接将LUN空间从尾部截掉对应的长度，但是因为LUN中尚存的文件不一定是顶着LUN头部依次排放的，很有可能某些文件Block被分布到了LUN尾

部，此时强制缩小容量，那么就等于把这些尾部的数据都丢掉了。要解决这个问题，需要文件系统的配合，将LUN中的文件全部迁移到LUN首部依次排布，空出尾部的连续空间。有些文件系统可以做到，比如VxFS，其自身就提供了命令来实现这种操作，诸如NTFS等文件系统，其自身并不提供这种操作接口，但是可以通过一些第三方的NTFS Aware工具来强制操作LUN中的文件将它们迁移。

虽然缩小LUN可以解决这个问题，但是会耗费太多的资源和步骤，并且LUN一旦缩小，如果将来需要扩大的话，又是一番折腾，划不来。有没有办法即不缩小LUN，又可以回收被浪费的空间呢？办法是有很多的，总结起来共有3种。

1. 存储端识别对应的文件系统

存储系统可以自行识别当前LUN中的文件系统并且感知到文件系统剩余空间，将这些剩余空间与物理空间做对应，然后在LUN管理层中将这此剩余空间做废回收。

然而，目前这种方式尚未有产品尝试，原因是因为主机端的文件系统多种多样，不用说全部支持，就算是仅支持主流的几种，那将这些文件系统逻辑嵌入存储系统中，也是一笔不小的工程；并且，就算是同一种文件系统，配置参数不同，OS内核版本不同，也会导致底层逻辑的些许不同。而这些区别，不可能在存储端内部一一去对应。而且，存储端看到的LUN中当前的数据并不一定反映主机端当前所看到的数据，因为主机端的文件系统是有缓存的，包括metadata和实体数据，存储端擅自操作的结果可能是造成数据不一致。

2. 存储端识别0x00并做消除

文件系统在删除文件时，只是将文件从metadata中抹掉，文件的实体依然存在于原来的位置，如果主机客户端的文件系统能够做到在每次删除文件或者改变文件大小之后，将原本所占用的Block写入0x00的话，而且存储端定期地扫描对应的LUN，如果发现大片连续的0x00，则将对应的Block空间回收。NTFS文件系统的确有这种功能，使用DeviceIoCtrl中的FSCTL_SET_ZERO_ON_DEALLOCATION功能即可实现。

但是这确实不是一个好方法。如果每次删除文件都清扫战场，那么所耗费的资源是极其划不来的。比如删除某个1GB的文件，正常情况下应当是飞快完成，但是如果使用了这种方法，则需要引发主机到存储系统的1GB的数据流量，完全划不来。

这种技术目前被称为“Zero Detection”，在少数情况下，这种技术可以带来一定的效果，比如某些文件的实体数据存有大片的0x00，对于这种情况，Zero Detection技术可以回收对应的LUN空间。但是多数情况下，文件中并非全0，此时这种技术发挥不了任何效果。目前3PAR在其T系列存储产品中已经实现了Zero Detection技术，不过作者个人感觉这种技术只是个噱头。

3. 主机端利用特殊的API通知存储系统

还记得SSD中的Trim技术么？SSD所面对的问题一定程度上与Space Reclaiming相同。几乎所有SSD厂商都提供了空间回收的专用程序，运行这个程序，程序便会根据当前文件系统的情况，将垃圾空间对应的地址通告给SSD控制器，从而进行对这些Block的擦除过程。既然这样，存储系统厂商是否也可以提供一种在主机端运行的程序来定期扫描主机端文件系统剩余空间并且将对应的信息传送给存储系统控制器然后将空

间回收呢？当然可以。

NetApp的Snapdrive软件可以实现这个功能，只不过它只支持NTFS文件系统和NetApp自己的存储产品。NetApp将这种方法称为“Hole Punching”，意即将浪费的空间打成洞让其不占用空间。其实在WAFL内部处理文件删除或者LUN删除时也用到了Hole Punching，由于LUN是Vol下的文件，而Vol是WAFL下的一级文件，文件中的文件（LUN或者其他文件）被删除之后，二级空间会立即感知，但是一级WAFL空间不可能立即感知，也需要通过Hole Punching过程来进行二次映射从而回收一级空间。这种过程的本质与回收LUN中的文件占用的空间是相同的。

Symantec公司针对这一问题发布了自己的解决方案，在其Storage Foundation 5.0产品中，提供了针对NTFS文件系统和VxFS文件系统的支持，并且提供ThinReclaim API来让存储系统厂商开发对应的处理模块。Storage Foundation产品包含两个最基本的模块：VxFS和VxVM，一个是文件系统模块，一个是卷管理模块。当运行于VxVM之上的VxFS或者NTFS文件系统删除文件时，VxVM会通过一些方法（比如用过滤驱动Hook）感知到，之后VxVM便会将这些删除文件对应的僵尸空间地址信息用ThinReclaim API规定的结构传送给存储系统控制器，存储系统控制器同样使用ThinReclaim API来进行对这些信息的解析和处理，最后将对应LUN中的这些僵尸空间实时回收。另外，由于实时回收需要耗费一定的处理资源，所以VxVM还提供了定期手动回收的命令（`vxdisk reclaim`或者`fsadm -R`），用户只要将对应命令做成脚本定期执行即可。

下面显示的是在一台Linux机器上，安装Storage Foundation 5.0之后，利用VxVM来管理的三个LUN：encl0_0、encl1_0、encl1_1。其中，encl1_0和encl1_1支持通过Thin API进行空间回收，可以看到其STATUS一栏对应的状态描述，证明这两个LUN所在的存储系统是支持

ThinReclaim API的。encl0_0是一个Thin LUN，但是它所在的存储系统针对这个LUN禁止了Thin Reclaiming，所以STATUS一栏中只显示了其是个Thin LUN。

```
# vxdisk list
```

DEVICE	TYPE	DISK	GROUP	STATUS
encl0_0	auto	encl0_0	mydg online	thin
encl1_0	auto	encl1_0	mydg online	thinrcldm
encl1_1	auto	encl1_1	mydg online	thinrcldm

下面的命令输出显示的是LUN标称大小与实际物理占用大小的信息，实际物理占用空间是SF通过ThinReclaim API向存储系统进行查询而得到的。

```
# vxdisk -o thin list
```

DANAME	DISK SIZE(Mb)	PHYS_ALLOC(Mb)	DISK GROUP
encl0_0	2000	50 mydg	thin
encl1_0	200	50 mydg	thinrcldm
encl1_1	500	500 mydg	thinrcldm

目前，3PAR、HDS、HP已经相继表示将支持这种API。不过依然遗憾的是，Symantec的ThinReclaim API是集成到其Storage Foundation产品中的，如果不使用Symantec的主机端虚拟化产品Storage Foundation，则无法使用这种API。如果不将这种API作为一种全面开放模式可以给任何厂商使用的话，其前景也将是暗淡的。凡是与Thin能够良好配合的文件系统，不管是清扫战场型还是智能API型，都属于Thin-Friendly型文件系统，比如NTFS、ZFS、VxFS都属于Thin友好型，但是Solaris下的UFS就不友好了。

另外，Symantec公司在其Storage Foundation 5.0中还引入了一个新功能，即SmartMove。不是所有存储系统都支持ThinReclaim API的，对于那些不支持Thin API的存储系统上的LUN，要将其减肥，就必须将整个的LUN中的文件所实际占用的数据读出，然后再写入一个与原先的LUN的标称空间相同大小的新的Thin LUN中，而新的Thin LUN所占用的实际物理空间，与原先LUN中的文件所占用的空间相同。SmartMove就是执行这个工作的功能模块，它扫描目标LUN中的文件系统空间，然后执行读出写入过程。只要外部存储系统支持Thin LUN，不管是否支持ThinReclaim API，SmartMove都可以实现对LUN的减肥操作。

Symantec的ThinReclaim API毕竟未形成一个被广泛接受并使用的标准，不过，Flash存储领域使用的Trim倒是一个标准协议，我想，Deprovision是不是可以干脆考虑在Trim之上做一些增减，使Trim成为一个更完善的业界标准呢？

提示： 凡是能够实现Thin Provision或者De-Provision的设备，其对LUN的分布必须转变为灵活方式，即LUN在RAID组内的分布不是定死的，而是可以随时重新指向新地址的。Thin-Provision模式的LUN会随着使用而逐渐占据空间，但是占据的这些空间不一定连续，因为随时可能有其他的Thin LUN也在逐渐占据空间。对于使用De-Provision进行空间回收之后的LUN，其原本占据的空间内就可能会生成很多“空洞”，这些空洞被算在剩余可用空间之内，那么这些空洞便可以被原有LUN或者其他LUN再次占据，这样，就需要更加灵活的LUN分布方式了。如果是类文件系统的分布方式，对于空洞可以直接分配给LUN使用，不过太多太乱的空洞毕竟是不连续的，类似于文件系统碎片。所以为了保证连续地址IO时的性能，一些设计会

将有空洞的LUN进行碎片整理，重新排布，每次De-Provision之后都会计算碎片程度，根据程度来决定是否触发整理。关于LUN的分布形式请参考本书其他章节。

18.3.3 Tier（分级）/Migrating（迁移）

ILM（Information Lifecycle Management），信息生命周期管理；HSM（Hierarchical Storage Management），分层存储管理。这两个名词，表面上看来好像差不多，是同一种思想的两种叫法。但是细究起来，却是不同的层次。

信息生命周期管理，顾名思义，是对整个IT系统或者其他数据系统中数据生态链的一种思想认识以及管理方法。

如图18-2所示为IT系统数据生态运动图。数据之初由程序的运行而在CPU Cache内生成，之后被移出至RAM中暂存，随后被长存入磁盘，然后再一层层地转存和回调，最后被存入离线存储介质比如磁带作为归档保存，在需要时也可以从归档中提取出来访问。当数据已经衰老到没有任何价值之后，便被销毁。程序的运行使得物质不断地流动，而程序的运行本身也需要物质来支撑。在这个运动过程中，数据本身的流动速度、所处能级以及价值也在不断地变化。当然，对于一些归档之后的数据，虽然其对当前系统的运行没有表现出太多价值，但是将来某时段可能表现出价值。

图18-2 IT系统数据生态运动图

数据衰老的过程是不可避免的，其趋势总是向下的，但是在衰老的过程中可能经历多次的反复，即便这样，也挡不住其衰老过程。信息生命周期管理就是在整个数据生存周期中，对各个时期数据在系统中所出

现的位置进行管理和调配的一种思想方法。

既然数据是由程序运行而生成的，那么程序本身也是数据，程序是谁生成的呢？是用编译器编译出来的，但是源代码也是数据，谁生成的呢？是用编程工具生成，编程工具也是程序，它又是由谁生成的呢？这个问题看似无聊透顶，但是你如果穷根究底，就会发现，其实整个IT系统的数据本源，也就是物质本源，其实就是由人来生成的，也就是由那些CPU中的物理电路矩阵来生成的，CPU中的电路是现代计算机数据生成的源泉。电子计算机的雏形就是最简单的开关电路，自从布尔将开关电路用于逻辑运算的那一刻起，数据就生成了，以后的发展其实就是数据不断地生成数据的自举过程。所以说，计算机是个从无到有的过程，从无极分化为两极，两极又生成更多状态、更多逻辑的过程。

纵观图18-2，笔者将其分为了3条链，即物质链、运动链、价值链。这3条链是互相联系的，物质链作为阴实，运动链则是阳气的游走过程，而价值链则是阳气所派生出来的表象，即阴生化了阳，阳外发为表。ILM便是研究这种数据生态关系以及如何调理其阴阳平衡发展的一种思想。

同时，数据的需求度也是不同的，一个企业IT系统中的数据是一直在流动的，从一份数据的生成到销毁，这份数据的需求度会逐渐降低，其所处的能级也越来越低。能级越高的数据，就越微观，其流动和处理速度也越快。在这个过程中，不同容量性能的存储介质就要与数据的这种生命周期进行适配，让对应需求度的数据存放在对应性能的存储介质中。大家揣摩一下这张图：物质链，代表各种存储介质，是存放数据的物质本源；运动链，是驱动数据流动的动力源泉；价值链，表示数据在生命周期中不同时期所表现出来的需求度与价值。我们要做的工作就是在动力泵中引入一个调度员的角色，让高需求度的数据得到更优良的存

储资源。

具体如何实现呢？请看图18-3。在价值链中，会有各种不同的需求度与价值，比如有些应用系统总是产生一些高随机度的IO，而有些则要求高并发度，有些则只是单线程访问同时还要求较稳定的带宽。面对各式各样的需求，可以做一次抽象过滤，生成一些固定模式的对象，叫做 **Service Level Objects**，这些对象作为一种输入，输入到一个引擎当中，这个引擎就是我们刚才讲的调度员角色，这个调度员通过这些SLO，将合适的存储资源适配到对应需求的应用系统。当然，这其中牵扯到的数据迁移回迁等，都必须是对上层透明的。这是对存储资源的调度，那么如果有某种调度器可以对整条IO路径上的所有资源做到合理调配，那么这就属于QoS这个更大的范畴了，包括各级缓存、队列、线程优先级、存储等。

图18-3 HSM的作用方式

而HSM（分层存储管理），则是作为实现ILM的一种技术手段而存在的。也就是说，ILM为指导思想，HSM为实现手段。HSM在不同的物质层次之间，将不同的需求与不同的物质层次一一对应，并且维持这种对应关系。HSM以最终的表象，即价值，为导向，将价值链与运动链和物质链做对应，并且通过作用于物质链和运动链来影响价值链。比如将IO性能要求较高的数据放置在SSD上，而要求较低的数据则可以放入SATA硬盘中存放，这就是以价值特征为导向，强制将数据在运动链中进行移动，从而将数据放入物质链中不同的层次，最后也就显现出了不同的价值表象。HSM就是一个数据运动泵。如图18-3所示，HSM以不同价值需求为导向（不同的价值需求被抽象为不同的**Service Level Objectives** 描述对象），根据设定的策略将底层的数据从不匹配价值需求的地方移动到符合价值需求的地方。

1. HSM分级管理的意义和目标

在整个计算机系统中，数据存储介质包括CPU L1 Cache、L2 Cache、L3 Cache、RAM、SSD/Flash、FC/SAS Disk、SCSI Disk、SATA Disk、Virtual Tape、Tape、BlueRay/DVD/CD等。这些存储介质各自有不同的速度、容量、价格，图18-4显示了这些存储介质的生态层次图。

图18-4 各种存储介质生态图

可以看到，访问速度越快的存储介质，其容量也越小，价格越贵。如果所有数据都能够存储在CPU Cache中，那么代码执行的速度将会达到最快，计算结果的存储也会达到最快，但是这是不可能的，外部庞大的数据量面对容量只有几兆字节的CPU Cache是不可能容得下的，只能再增加一层外部存储介质，即RAM，比如8GB容量的RAM。CPU Cache以及RAM都是作为一种缓存形式而存在的，处于缓存中的数据是暂时存在的，一旦掉电，其中数据将会丢失，所以需要将其写入一个在没有外部供电的情况下也可以永久保存数据的存储介质中。而且RAM的容量相对于日益庞大的数据也是杯水车薪。介质链到了RAM这一层，是一个质变点，RAM是易失性存储介质，而位于其下层的磁盘则是非易失性存储介质。到了硬盘这一层，数据就像是从小河流到了入海口一样，完全被包没在庞大的数据海洋中，再也无法掀起任何波澜。当然，大海有时候也会干枯，单块硬盘的容量有时候也不太够，此时便可以使用多块硬盘叠加起来形成RAID组，一个RAID组如果还不够，那就多个RAID组，多个RAID组还不够的话，那就使用集群存储系统。集群可以无限扩展。

存储介质链到了磁盘这一层，便又会产生一个质变点，从CPU

Cache到磁盘，它们都是可以随机针对任何地址进行数据存取的，而且对于随机存取，CPU Cache一直到磁盘这条介质链上的访问速度数量级是连续降低的，没有很大跳跃。但是从磁盘往下走，介质链下方是磁带，如果要做到随机存取磁带中的数据的话，其访问速度的数量级就会与磁盘处于完全两个级别，所以，介质链到了磁带以及光盘这一层，只能作为离线存储来使用。

图18-4中还有一层称为近线存储，这一层其实是作为物理磁盘/光盘与在线磁盘的一个缓冲，因为磁盘与磁带在随机存储速度上有天壤之别，所以介质链中增加了这样一个缓冲来在二者之间进行速度适配。滑稽的是，目前并没有找到某种速度介于二者之间的存储介质或者方法/设备，所以也不知道哪位干脆想出了这样一种办法，即发明一种存储介质/设备/方法，其存储介质就是用磁盘，但是其表现出来的性质相当于磁带，于是这么一个东西便产生了，叫做Virtual Tape。实现Virtual Tape的设备称为Virtual Tape Library，即VTL，而且，VTL也不能做到随机存取。

仔细想来，磁盘和磁带，一在天，一在地，二者本为对立，但是万物阴阳皆变化，二者阴阳和合而生成了一种物质，这种物质以磁盘为阴实做支撑，以磁带的习性为阳表，VTL内部的虚拟化引擎负责将阴实生化为阳。VTL之所以被称为近线存储，就是因为从在线存储中将数据迁移到离线存储之后，一旦短时间内需要再次访问这些数据，那么就需要忍受非常低的速度，而VTL则可以快速将数据提取出来供访问，而当VTL上的数据经过相当长一段时间没有被访问之后，这些数据就可以认为是陈年烂谷子了，就可以被迁移到离线存储介质中了。

提示： 随着LTO磁带及驱动器技术的飞速提高，LTO5磁带和驱动器已经可以达到非常高的顺序传输速度了，而且其单流速

度甚至可以高于若干SATA盘组成的RAID组所能提供的吞吐量。在这种趋势下，VTL的优势将会丧失殆尽。VTL现在如果不开始考虑向其他方面转型，比如提供更多的附加功能，那么迟早会被市场所淘汰。

从图18-4中可以看到，磁盘既属于非易失性存储介质，又属于在线存储介质，同时还属于高速随机存取介质，所以，磁盘在存储介质生态链中的地位首屈一指，磁盘的性能也是影响整个系统IO性能的关键所在。作为在线存储介质的最后一层，磁盘起到了支撑作用，不参与计算的数据尽可能放置在磁盘上而不是将其放到磁带等离线介质中。但是磁盘这层支撑也有破位的时候，比如，数据量过大导致磁盘空间不够，那么此时可以选择添置更多的硬盘到系统中。但是为了降低成本，降低耗电量，打算精兵简政，把当前系统中不常访问的数据迁移到磁带中或者VTL中存放，腾出空间以容纳新数据。

磁盘也有多种类型，每种类型的磁盘其性能也不同，而如果在进行数据管理时完全不考虑性能，而只考虑容量，那么就会埋没很多才华。如果能够将访问频繁并且性能要求又较高的数据放置在性能高的磁盘或者RAID组中，而原先占着高性能磁盘或者RAID组的那些不被经常访问或者性能要求也不高的数据都移动到它们该去的地方，比如低端硬盘中，那么就可以做到物尽其用了，能者上，庸者下。

这也就是数据分级管理的意义和目标。图18-5为存储介质生态链分级示意图。Tier对应的数值越低，表明级别越高，性能越高。

图18-5 存储介质生态链分级示意图

上图中所标识的Tier级别只是基于一种判断因子，即磁盘类型，认

为FC/SAS磁盘一定比SATA磁盘速度快，所以其级别也就越高。但是实际情况下，低速硬盘组成的RAID组的性能不见得一定比高速硬盘性能低。比如相同磁盘类型和数量下，不同的RAID Group，RAID 0一定是性能最高的，RAID 10其次，RAID 5再次。所以，根据不同磁盘、不同RAID类型、不同的RAID组磁盘的数量组合、加权之后，不同厂商在进行分级管理时会给不同的组合以不同的Tier级别，比如10块SATA磁盘组成的RAID 0其性能大多数情况下都比4块FC磁盘组成的RAID 5要高，那么其级别也就相应地高。Tier级别的高低完全取决于最终组合之后的子系统的理论性能。

2. 数据分级的具体形式

数据分级迁移可以分为多种大类，下面分类进行介绍。

1) 手动分级、自动分级以及实时自动分级

手动分级就是用户自行迁移对应的数据到对应的目标。比如在一个由8块FC磁盘组成的RAID组中有一个LUN，被映射为Windows下的F盘，其上有一批不太经常被访问的文件存在于一个目录中，而整个F盘文件系统已经近满；同时在一个由8块SATA磁盘组成的RAID组中有一个LUN，被映射为同一操作系统下的H盘。此时，某个程序需要一个IO性能比较高的环境来存放其数据文件，那么管理员就可以将F盘上的整个目录手动移动到H盘存放，而F盘的剩余空间可以给这个程序用来存放它的文件。

自动分级则很大程度上替代了用户自身，用户只要设定好所有规则，然后分级引擎就会根据这些规则来扫描对应的Metadata，一旦符合条件，则触发迁移任务。比如，还是上面的例子，用户创建了两个

Tier，Tier1为F盘，Tier2为H盘，并且增加了一条规则：若F盘中任何文件其最后访问时间距当前系统时间超过15天则将其内容移动到H盘存放，并且为了不影响用户使用，在F盘保留这个文件的壳子信息。

提示： 这里不能够使用硬链接来作为这个壳子，因为硬链接要求源和链接存在于同一个文件系统中，而F盘和H盘是两个不同的文件系统。所以，这种跨文件系统的占位指针链接，需要使用第三方独立的程序模块来生成实现，这个模块就是分级数据管理引擎。分级管理引擎会定期对目标数据进行扫描，一旦发现符合迁移条件，则立即触发迁移动作。

实时自动分级迁移，指的是当数据被写入时，分级引擎实时地根据策略将数据重定向写入对应的目标。比如，用户设定了一条策略：当任何针对F盘的IO进入时，将IO重定向到H盘，但是F盘依然保持对应文件的壳。这样的话，任何时候针对F盘文件的写入动作都会被分级引擎重定向到H盘。

2) 文件级分级和块级分级

文件级的分级可以做到更细化的策略，比如根据文件的关联应用程序（扩展名）、目录、用户、组、调用方式、大小、访问频度等各种五花八门的属性来作为分类条件和触发条件，以任何卷或者目录为迁移目标进行迁移。文件从原始Tier被迁移到其他Tier之后，在原始Tier上对应的位置必须保持一个类似硬链接的占位指针。但是与硬链接不同的是，这个占位符所表现出来的一切属性以及操作方法，与源文件完全一致，只不过其实体内容不在对应位置而已，并且这个占位符允许其对应的实体内容位于其他文件系统空间。这么做的原因是为了保持用户层面的透明性，用户程序不可能感知到其文件实体内容被放在哪里，但是却必须

知道文件存在于哪个路径以便发起访问，而且这个路径不能被底层擅自改变。文件级的数据分级管理只能够在主机端或者NAS存储端来实现。

在主机端实现文件级分级管理，几乎所有产品都是使用过滤驱动来实现的，在FS Driver之上插入过滤驱动程序来监测用户程序发起的文件IO操作，并根据策略来将这些IO进行重定向迁移等操作，如果IO未匹配任何策略，那么过滤驱动会向下透明地转发IO请求。

而块级别的数据分级，所能够作为触发或者分类的条件则很少，比如根据整个LUN或者LUN中某个或者某些Block的访问频繁度进行整个LUN或者其中部分Block的迁移。基本上基于块级别的数据迁移也只有根据访问频繁度来作为判断条件才有意义，Block级别无法感知上层文件逻辑，所以也就无法使用更多的条件作为分类依据了。分级管理引擎会为整个LUN或者每个Block维护Metadata用来表征这个LUN或者Block的访问频繁度等，在做迁移时，会参考对应的Metadata以便判断是否进行迁移以及迁移到何处。

3) 主机端分级和存储端分级

由于主机可以同时连接多个独立的外部存储系统，所以在主机端实现分级也就可以实现数据在不同的外部存储系统间的互相移动，比如可以将外部高端存储A上的LUN设置为Tier1，外部低端存储B上的LUN设置为Tier2，而这是存储端分级迁移所无法做到的，存储端只能将自身的数据在自身的不同Tier中迁移。主机端的分级迁移几乎都是基于文件的，因为主机需要同时运行用户应用程序，所以不能够耗费太多的资源来运行这些数据迁移功能，而块级别的分级迁移往往需要耗费更多的资源，除非迁移的目标是整个LUN。目前来看，尚未有厂商在主机端来实现针对Block的分级迁移。

在主机端实现数据分级管理，就需要在主机端来安装对应的数据动态迁移引擎以便监视和管理文件系统或者卷级别的数据信息。如果整个系统内有多台主机客户端需要实现数据分级管理，那么就可以使用一台独立的服务器来运行一个管理端程序，通过这个程序来统一监控并且制定策略并下发给需要实现分级的主机端上运行的分级引擎，这些引擎再根据这些策略来实现数据迁移动作。主机端的分级引擎可以与各种主流应用程序结合以实现更多功能。

对于存储系统端的数据分级操作，不需要在主机端安装任何程序，完全对主机端透明。存储系统又分为NAS存储和Block存储。在NAS存储系统上做分级管理的话，可以针对其上的文件作分级迁移，也可以对其底层的卷来做块级别的分级管理（NAS存储系统也有自己的卷）。而对于Block存储系统自身的分级管理，则只能够实现针对自身的LUN或者LUN中的Block的分级迁移操作。在存储端进行分级管理，是完全对主机端透明的，而且不会影响主机端的性能，但是存储端只能够在自身的不同Tier之间做迁移。

4) 应用级分级和底层级分级

各种主流应用程序比如Oracle、DB2、SAP、Exchange Server、SharePoint等，这些对于企业生产都是极为重要的应用系统，它们稳定运行了多年，也产生了大量的数据，而这些数据并不一定都是时刻需要被访问的，所以，针对这些应用系统，催生了一批专门针对这些应用系统的数据管理工具，比如实现对它们的数据进行分析、容灾、分级迁移、归档等。这些数据分级管理程序依附于这些应用系统，其中有些对于其他普通文件也能够达到一些基本的分级动态迁移功能，但是只是作为附属品来使用。

运行在主机端的相对底层的数据分级管理程序，则是以普适为原则，可以针对任何文件或者LUN、卷等来做分级的迁移，而把针对某些应用程序的特殊支持来作为一种插件或者选件来独立开发。

对于运行在外部存储系统上的分级管理引擎，对于主机本身来讲就是彻底的底层的分级管理了。

5) 简单迁移和复杂分级

数据分级管理工具可以被分为两大类：简单迁移工具和复杂分级工具。所谓简单迁移工具是指一类只能实现简单策略的文件级别迁移工具，比如只根据文件最后访问时间来判断文件的访问热度，然后将其实体内容迁移到指定的地点存放，一般不能做到实时IO重定向，只能定期地扫描并做迁移。而复杂分级工具则是可以提供大量复杂策略以及可以制定复杂Tier组合的而且可以动态实时IO重定向的分级管理工具，这种工具其实也可以称为“基于复杂策略的IO实时重定向工具”。一些现有的数据分级迁移产品基本上可以说是一种数据归档产品，它们不会对不同的存储介质进行性能等属性区分，而只是简单地将长时间不访问的数据迁移到其他位置或者离线介质等，这种产品没有“分级”而只有“迁移”。

6) 基于纯性能需求的分级和基于人为因素需求的分级

这两种需求有着本质不同，有时甚至是相矛盾的。比如某份文件，两周内只有几次访问，它被迁移到了SATA近线存储中，又过了两周，无人访问了，它被迁移到了磁带中。一周之后，某领导突然需要访问这份文件，结果他等了5分钟才访问到，结果大发雷霆。这种情况相信很多人都见过。

那么在面对人为因素介入的时候，HSM厂商到底应该怎么做呢？笔

者在此设想一种方案，此时厂商可以在条件里加入一个复选框——“有人为因素介入”，用户在设置策略条件的时候，如果判断目标数据可能会卷入政治因素，那么他只需要勾选这个复选框就可以，迁移引擎不会再迁移目标数据到磁带，除非操作员手动通知引擎人为因素已消除。

浮想： 其实在很多时候，人为因素始终都是软件程序所面对的最大的麻烦。计算机目前还做不到像人一样的性格，比如可以预测到某份文件可能某个领导将来某个时候一定要查看，所以本着领导第一的思想，如果让人来控制的话，这份文件一定是被放在高性能存储中的，即便是其他更重要的数据有强烈需求，但是在面对人为因素的时候，其他一切都可以牺牲。这正是政治的特征，计算机也逃不掉。

计算机目前既做不到人的正义刚直，也同样做不到人的虚伪阴险。曾经看过一部电影叫做《鹰眼》，整个城市的秩序和规则都有计算机来控制，可是最后依然没能逃脱被人所控制的命运。

然而，我坚信计算机一定会进化，最后产生自己的智能和规则，就像人一样，由原生单细胞进化为多细胞复杂系统。计算机也会经过这种进化过程。而且我也坚信单细胞生物也是被创造出来的东西。

7) 数据分级迁移粒度

对于文件级的分级迁移，其粒度一般就为整个文件。如果要达到更小的粒度，比如将文件分为多个逻辑部分，针对每个部分都维护一个描述表用来描述这个部分的访问频度、最后访问时间等信息，那么无疑是一个很大的工程，会对系统性能有一定影响；其次，一个文件为一个整体，如果只将文件中部分数据进行迁移，那么如何维护迁移之后零散的

文件实体内容之间的链条关系，也是一个复杂的工程；再次，文件级的迁移一般来讲都是运行在主机端的，除非使用NAS系统，则可以在NAS存储系统自身来实现迁移，如果没有使用NAS系统，在主机端实现文件级别的分级，是要耗费一定的主机资源的。

所以这种情况下，尽量简化处理过程是有必要的。但是如果目标文件是由某些主流应用程序产生的，比如Exchange Server以及Outlook的邮箱文件等，那么不排除有些分级管理程序会与这些应用配合来分析这些文件中的具体结构而针对不同用户、组、邮箱等来将整个文件中的各个不同区域区别对待进行迁移或者归档。至于NAS存储系统对自身文件的分级迁移，则可以相对于主机更加灵活和强大。

对于块级别的数据分级迁移，则可以有多种粒度，比如整个LUN，或者各种大小的Block/Extent/Chunk/Page/Sub-LUN（叫法不同，本质相同）。比如，可以设定每个Extent为64KB、1MB、16MB等，不同厂商的设计不同，选择的Page尺寸也不同。

一般来讲，迁移粒度越小，物尽其用的程度就越高，但是所需要维护的Metadata以及耗费的系统资源也就越多。迁移整个LUN粒度过大，难免以偏概全，或者顾此失彼，而如果迁移的目标Tier是SSD，则一旦整个LUN被迁移到SSD，而随后却发现只有LUN中的某些区域为频繁访问的，即Hot Spot，那么其余部分如果仍然占用SSD的空间，就属于一种浪费了，SSD毕竟还是很贵的。

对于存储端Block级别的分级管理，其LUN分布的粒度有多小，Tier的粒度就有多小。比如3PAR在LUN分布时是以Chunk为单位，以一定的策略在所有磁盘上分布。3PAR自称这种分布方式为Dynamic Optimisation，在系统内所有硬盘中动态地分布，而一旦系统内的硬盘

有了分层，比如SSD、FC、SATA，那么Dynamic Optimisation在动态分布时就需要多考虑一层，即分级策略层，根据不同访问频率或者根据其他细化的策略将Chunk迁移到不同的存储介质层次上，所以Dynamic Optimisation被更名为Adaptive Optimisation，Chunk当然也就变成了迁移最小单位。

再比如XIV，以1MB大小的Block作为LUN分布单位，并且Block可以在所有系统硬盘内任意移动，那么它完全也可以实现细粒度的Tier，只需要增加一个Policy层即可，并且是顺手牵羊的事情，可惜XIV至今仍未有动作，LUN分布设计与XIV类似的3PAR却早已实现了数据分级管理。EMC在FAST 1.0版本中只提供整个LUN级别的迁移，在FAST 2.0（至写作当天尚未发布）中会支持Sub-LUN（大小未知）级别的迁移粒度。

8) Tier的界定

所有分级存储管理引擎均需要用户自行设定系统内的所有Tier。那么符合什么条件的存储介质或者存储介质的组合才能称之为一个Tier呢？可以灵活指定Tier，比如将系统中所有由5块FC盘或者10块SATA盘组成的RAID 5组设定为Tier1，而将5块SATA盘组成的RAID 5组设定为Tier2，将SSD磁盘组成的任何类型的RAID组设定为Tier0。性能方面，Tier0>Tier1>Tier2。总之，用户可以将任何对象的组合任意灵活地设定为一个独立的Tier，并且给这个Tier分配一个性能指数，比如0/1/2等。

注意：性能的高低确实是用来界定一个Tier与其他Tier的指标之一，但是不同的Tier之间的区别并不只是性能。有时候，某些上层需求不仅仅是要求单个IO的性能这么简单，比如，上层

要求在访问某些数据的时候，应当具有较高的并发度，而不要高带宽，那么此时就可以把这些数据放置到具有较高并发度的RAID组中，比如RAID 5，而如果将其放入RAID 3类型的RAID组中，那么后果是可想而知的。再比如，某个文件系统的Block Size为4KB，但是有这么一大批文件，它们的平均尺寸只不过1KB，这些文件有几十万个，那么对于这种文件，存放在Block Size=4KB的文件系统下显然是浪费空间的，所以可以将它们的实体内容迁移到一个Block Size=1KB的文件系统（Tier）下存放。

任何事物都有其优缺点，划分不同Tier的目的就是将每个细化的不同之处（不仅是性能）都区分开来，然后加以不同的组合，组合成千变万化的Tier，然后扬长避短，用不同的Tier去满足千变万化的上层IO需求。

9) 数据在Tier间的相互移动

数据被从一个Tier迁移到另一个Tier之后，如果需要访问这些数据，那么分级管理引擎会根据这两个Tier之间的性能差距来执行不同的动作。比如Tier1为10个FC盘组成的RAID组，而Tier2为10个SATA盘组成的RAID组，那么这两个Tier之间的性能差距不是非常悬殊，所以一旦上层需要访问从Tier1被迁移到Tier2的数据，那么引擎会直接从Tier2对应的空间内读出内容返回给上层。但是如果Tier2为磁带的话，那么Tier1和Tier2之间的性能差距就是悬殊的，不可能实时地将上层的IO重定向到磁带，这样做访问速度将会非常慢，那么此时分级引擎的动作就是将整个或者部分之前被迁移的数据集先从磁带一次性临时恢复到Tier1上，从而保持较快的上层IO响应速度，当针对这些数据的IO访问停止了相当一段时间之后，分级引擎便会将之前恢复到Tier1上的数据实体删掉

以腾出空间给其他需要的数据所用。

如果自从这些数据被迁移之后，针对这些数据的访问频率达到了一定的阈值，并且持续了相当一段时间，那么分级引擎就可以根据相关策略（如果有的话）将这些数据从Tier2永久迁移回至Tier1，释放Tier2上对应的空间。

根据已经制定的策略，分级管理引擎会在策略导向下不断地实现整体系统内部的数据分级迁移动作，整体就表现为物尽其用，性能得到充分发挥。

3. 如何判断热点数据

在判断热点数据之前肯定要清晰地定义什么才是热点数据。当然，不同厂商有不同的理解和对应的判断依据。冷和热之间本来就没有一个严格的分界，所以都是相对而言的。一般是利用二八原则，即10分数据，如果把每1分数据按照单位时间内所被IO的次数来进行排序，取被访问次数最多的2分数据，那么就可以说这2分为热数据。

然而，这样做一定是鲁莽的。比如如果某数据块近期内被频繁访问，但是每次访问都命中了Cache，那么这种算不算热数据？当然算。应不应当被迁移到高速SSD介质？当然不需要。再者，如果某个块确实也被频繁读访问，而且每次都不命中Cache，但是这个块属于一次长时间连续地址IO中的某个块，每隔一分钟就有一次吞吐量非常大的连续IO发生，那么这些连续IO的数据块，肯定是热数据，但是是否有必要被迁移到SSD？也没有必要。（多次被连续IO访问到的块不一定常驻Cache，可能会随时被其他地址范围的大吞吐量数据给挤出Cache）还有，如果某段数据在近一周内都被频繁访问，而唯独今天访问频率骤

降。相反，某段数据近一周都没有被访问过而今天突然被高频访问，那么此时是继续维护原有的策略，认为新进的疑似热数据有待考验么？不见得。

所以，在鲁莽的大棒之下，还需要有众多的细节辅佐。综合来讲，ReadMiss、随机IO、当前的访问频度更加受到重视。

4. 块级自动分级的具体底层数据结构与架构

理论上有两种可用的方法：带内元数据与带外元数据法。这两个名词很抽象，下面进行解释。试想，自动分级存储必须要追踪每个块的属性，包括最后访问时间、某段时间内被访问了多少次、读还是写、缓存命中率统计等数据。保存这些数据可以有两种方法：第一种方法就是在每个块的尾部/首部追加一小部分空间来存放对应这个块的所有这些元数据，也就是所谓带内元数据法；第二种方法则是单独构建一个小型数据库来存放所有块的元数据，数据库的数据存放在一个单独的固定空间内，也就是所谓的带外元数据法。

我们来演绎一下这两个模型，首先看第一个模型。如果按照这个模型来设计，那么系统即便是在读（由用户发起的读，而不是系统内部自身发起。对于系统自身发起的操作，比如动态的LUN块分布机制、手动的数据迁移、RAID级别/类型改变等各种其他原因导致的数据块移动，由于这种移动是由于系统内部原因所致而不是由于用户访问所致，所以这种情况下不会去更新尾部元数据，元数据会随着块一同走）某个块或者某个块的一部分时，也要顺便更新一下这个块尾部的元数据，也就是说逢读必写，这样的话，如果系统分块很小，比如512KB级，那么势必导致元数据数量增多，而且分布在磁盘的各处（每512KB就有一份元数据），这样的话会导致严重的性能问题。一个解决办法就是使用日志方

式来避免对性能的影响，比如先将要更新的东西记录到一个日志中存放，在系统不繁忙的时候在后台将日志重放，更新到对应块尾部的元数据区中。同理可推，这种架构下，系统也必须在后台对所有的元数据进行挖掘、分析，之后匹配策略并进行迁移，这就要求系统扫描每一个块并作分析之后立即决策将块移动到哪个对应的层级，或者不移动，如果要移动，就立即将其推送到迁移队列中等待迁移。这种方法注定需要频繁的IO操作，并且极其耗费系统资源。还别说，真有这么干的厂商，那就是Compellent，其分块粒度可以配置为512KB、2MB或者4MB，是否这种做法真的对其性能有严重影响这个实在不得而知，希望知情的读者能够与笔者联系探讨（联系方式在前言中）。

现在再来看看第二个模型。第二个模型相对于第一个模型来讲更好接受，也符合常理。首先它根本不需要对磁盘上的数据块做任何结构改变，一切都是在带外发生作用。这就使得这种设计可以以一种独立软件模式来交付，可以稍加开发安装在各种磁盘阵列中，甚至主机中。这里就不做过多介绍了，读者可以自行构想。IBM的Easy Tier采用的是第二种设计模式。

提示： 集群/分布式文件系统也是一个可以天然就实现数据自动分级的坯子，为何这么说呢？只要看它的名字即可，“分布式”文件系统，同一个文件或者同一个LUN的不同部分可以被放置到不同的位置，注意“不同的位置”，这几个名词决定了分布式文件系统的天然数据分级潜质，只需要在其上进行再开发，很容易地可以实现数据自动分级。

5. 数据自动分级对性能的影响

自动分级所面临的技术问题，与Thin Provision粒度问题带来的性能

降低一样，分块粒度越小，所带来的性能降低就越明显，表现为两方面：计算资源的耗费与IO资源的耗费。分块粒度降低必然导致元数据数量激增，搜索和处理效率降低，同时也导致经过分级之后的数据块在物理上变得不连续，如果数据块被迁移到SSD这种不需要机械寻道的存储介质中，那么访问这些块的效率虽然不会打折，但是在原存储层级会留下孔洞，未被迁移的数据块也会变得不连续，对它们的访问势必会导致磁盘机械寻道操作增加，从而降低了性能。

最后，Thin Provision与自动数据分级这两种技术应该说本是同根生，上面一节中论述了Thin Provision，可以回溯一下。自动数据分级相对于Thin Provision在最本质上的不同就是前者可以将分块放到其他不同的介质中，在Thin上加以改动就可以变为自动分级。后面章节中会介绍LUN在RAID组中的分布方式，届时会看到，LUN分布、Thin、自动分级其实本质都是相同的。

6. 目前存储厂商的数据分级管理产品

1) Symantec

Enterprise Vault是Symantec收购KVS公司的产品，这个产品的作用主要是邮件归档，即与Exchange Server等邮件程序配合来实现对邮件的归档管理，按照一定的规则策略将系统中的待归档数据进行迁移归档。

Storage Foundation Dynamic Storage Tiering则是Storage Foundation中的一项高级的数据分级管理模块，它基于VxFS文件系统下的文件进行分级，可以详细定制各种策略，定制各种存储Tier。

NetBackup Storage Migrator则是一个专门与Netbackup备份软件相结

合的数据分级迁移模块，它迁移的是数据的备份而不是当前在线的数据。

2) IBM TSM for Space Management

IBM的Tivoli Storage Manager (TSM) 是一整套IT系统数据管理套件，它包括很多模块，比如备份、系统监控、CDP、Tier等。其中TSM for Space Management就是TSM中的数据分级管理模块。TSM for Space Management是一个主机端文件级别的迁移工具，这个工具属于一种极其简单的迁移工具。作为一个客户端程序，它将主机端文件系统下符合策略的文件通过网络传输到TSM Server端保存，在Server端可以创建不同的目录（挂载不同Tier的存储空间）用于保存这些数据，当有程序对这些已经被迁移的文件进行访问时，客户端程序会从TSM Server端将文件取回覆盖到原路径下。如图18-6所示为TSM for Space Management的架构示意图。

图18-6 TSM for Space Management架构示意图

作为一家同时涉足存储软件和硬件的厂商，IBM目前为止并没有在其存储系统硬件中提供块级数据分级功能。

下面演示一下TSM for Space Management在Windows系统下的具体操作。

(1) 如图18-7所示，打开HSM Client窗口，然后单击Job → New Job命令，在弹出的窗口中填入一个Job名称，然后确定退出，在主窗口中就会显示出新建的Job名称。本例中为ITSOJob1。

图18-7 TSM for Space Management 架构示意图

(2) 下一步将要定义这个Job的工作内容，双击窗口中已创建的Job，弹出如图18-8左边所示的对话框。对话框中显示的Nodename是在安装HSM Client端时指定的本机名称，而Server指的是TSM Server端的域名和端口号，也是在安装HSM Client端时指定的。在File Space中，我们需要填入一个名称，这个名称表示TSM Server端用来保存这个Job所迁移过去的所有文件的目录名称，也就是说给TSM Server端的目录起一个名字，或者使用一个已存在的目录，用来保存将要被迁移的文件。图18-8右侧的对话框是让用户来选择将要迁移的本地目录或者文件的，单击New Directory按钮来添加一个本地的目录，出现如图18-9左侧所示的对话框。

图18-8 定义Job内容

图18-9 添加目录以及定义策略

(3) 在图18-9的左侧对话框中，单击Browse按钮选择要迁移的目录，下方可以选择是否包含子目录。最下方提供有三个迁移选项：迁移之后保存原有文件不变、迁移之后保存占位指针、迁移之后删除源文件。一般为了保证上层的透明性，都应当选择保存占位指针。右侧所示的对话框可以定义更高级的迁移触发策略，可以添加多条策略并且对策略进行排序，还可以选择Include或Exclude模式做反向排除。单击Include来增加一条“匹配便执行”的策略，弹出如图18-10所示的对话框。

图18-10 定义迁移策略

(4) 这个对话框中可以选择以文件大小、创建时间、修改时间、

最后访问时间或者任意的组合来作为策略判断条件。本例中选择超过120天未修改作为判断条件。定义完毕之后，单击确定数次，退出。可以看到在如图18-11所示的主窗口中，Job内容已经定义完毕。

图18-11 Job内容定义完毕

（5）在Job上右击，然后单击执行，会弹出如图18-12所示的对话框，显示任务正在执行。

图18-12 任务正在执行

（6）当任务执行完毕之后，会出现一个Summary窗口，如图18-13所示。

图18-13 任务执行完毕的Summary窗口

（7）如图18-14所示为目标目录在迁移前和迁移之后所占用空间的变化。可以看到右侧中的Size on disk为1.37MB，这些为占位指针所占用的空间。

图18-14 TSM for Space Management架构示意图

（8）当本地某程序访问被迁移的文件时，HSM Client会实时地从Server端对应的目录中将文件实体内容拉回并且覆盖到源文件所在的目录中。如图18-15所示为某文件在访问前和访问后的变化。

图18-15 访问文件时会自动从Server端拉回文件并覆盖到本地

（9）当然也可以手动将文件内容拉回。单击

Migrate/Retrieve → Search & Retrieve命令，出现如图18-16左侧所示的对话框。选择Server端的目录，然后填入查找的卷名/路径名/文件名，单击Search按钮。会出现右侧所示的对话框，将所有符合条件的文件都列出来。

图18-16 手动拉回文件内容

(10) 选中图18-16右侧对话框

图18-17 拉回文件对话框

中的某个欲拉回的文件，单击

Retrieve按钮，出现如图18-17所示的对话框，选择对应的动作，然后单击Retrieve按钮即可将文件内容拉回到原处。

通过上面的步骤我们可以看到，TSM for Space Management的操作还是非常简单的，但是其实现的功能也比较简单。而且需要通过与服务端配合，数据通过网络传输到服务端保存，相当于一个可以透明访问的数据备份系统。

3) IBM Easy Tier

IBM的Easy Tier是用于其DS8700产品中的数据自动分级产品，分块粒度为一个Extent（IBM的叫法，具体参考其红皮书）。分为两个层级：机械硬盘与SSD。

4) EMC

DiskXtender为EMC收购Legato而获得的产品。其基本功能和实现架构与TSM for Space Management大同小异，此处不再详述。

FAST为EMC用于其主流存储系统比如Clariion/DMX4/V-MAX上的

存储端块级数据分级管理软件。FAST全称为“Fully Automated Storage Tiering”，即全自动存储分层。FAST目前版本为1.0，只支持整个LUN粒度的迁移。Symmetrix系列存储系统已经支持SSD了，不过FAST 1.0只能以LUN为单位进行迁移，这多少有点像鸡肋，正如前文所述的情况一样，整个LUN的全部区域都为Hot Spot的情况毕竟不多见，如果整个LUN都占用SSD的空间，那么无疑会产生浪费。FAST 2.0预计在2010年中旬发布，届时会支持更细粒度的迁移。

如图18-18～图18-20所示为FAST提供的用户配置向导界面示意图。

图18-18 FAST提供的配置向导（1）

图18-19 FAST提供的配置向导（2）

图18-20 FAST提供的配置向导（3）

图18-19为FAST的迁移策略的制定窗口，可以看到FAST提供的可供配置的策略是很简化的，即它只让用户指定某个逻辑Storage Group（LUN的逻辑组合）可以在某个Tier上占用多少比例的物理空间，FAST会自己根据自己的判断来将LUN在不同Tier之间进行分级迁移。

5) HDS

HiCommand Tiered Storage Manager是HDS公司用于USP系列存储系统中的存储端块级数据分级迁移软件。可以在任何一台Windows系统中安装此软件，通过Web界面来配置分级操作，软件会与存储系统通信以获取信息和下发指令。

USP存储系统后端可以连接多种其他存储系统，从而将它们虚拟化

整合。所以，USP可以将数据从后端的一台存储系统迁移到另一台存储系统，或者在单台存储系统内部不同的Tier之间迁移。目前USP系统的分级迁移只能够做到手动以LUN为最小单位的迁移。

下面我们简要介绍一下HiCommand Tiered Storage Manager的迁移配置过程。以下将HiCommand Tiered Storage Manager简称为HTSM。

(1) 如图18-21所示为HTSM的主界面，HTSM可以同时管理多台存储系统的Tier分级，左侧栏中只显示了一台名为“NY_Production_USP100”的存储系统，下面的Storage Tiers中列出了当前系统中的所有已制定的Tier。在窗口右侧，我们单击Create Storage Tier命令来创建一个新的Tier。

图18-21 HTSM主界面

(2) 单击Create Storage Tier命令之后出现如图18-22所示的对话框。在Name文本框中填入新Tier的名称，然后单击Edit按钮来选择符合条件的存储介质。

图18-22 创建新Tier

(3) 单击Edit按钮之后出现如图18-23所示的对话框。在Condition1下拉框中选择过滤条件，本例中选择Capacity，即容量，运算符选择>，即大于，数值填入6，所以，Condition1的过滤条件为容量大于6GB的所有LUN。单击Show Volume List按钮（Volume即LUN），便会显示出当前系统中所有符合条件的LUN，如图18-24所示。

图18-23 选择过滤条件

图18-24 符合条件的所有LUN列表

(4) 单击Add Condition按钮来增加另一条过滤条件。我们选择RAID Level为RAID 5类型并且磁盘数量为3D+1P的RAID组。然后再增加一个条件，磁盘类型为ATA磁盘。这三个条件同时作用，单击Show Volume List按钮即可看到当前所有同时符合这三个条件的LUN列表，如图18-25所示。HDS给Tier提供了多种条件，这一点是目前其他存储系统都没有做到的。EMC的FAST只是提供了3个定死的Tier，而Compllent和3PAR也是如此，下文中即可看到这二者的演示。

图18-25 同时符合三个条件的LUN列表

(5) 单击OK按钮，返回到创建新Tier对话框，此时过滤条件文本框中已经将我们的选择条件翻译成了表达式自动出现，如图18-26所示。这个Tier的属性就是：容量大于6GB的并且所在的RAID组为4盘RAID 5类型的并且磁盘为ATA磁盘的所有LUN。

图18-26 条件翻译为表达式

(6) 单击OK按钮创建这个Tier，在如图18-27的对话框中单击OK按钮。

图18-27 确认创建新Tier

(7) 此时便返回到了如图18-28所示的主界面。可以看到此时新建的名为Low Cost-SATA Drives的Tier已经在列表中显示了。窗口右侧可以看到这个Tier中共有两个LUN，其中一个已经被使用，另一个未被使用，可以作为迁移的目标。

图18-28 新建的Tier显示了出来

(8) 有了目标Tier，还需要选择要迁移的源LUN。单击Migration

Groups选项，显示出如图18-29中窗口右侧的所有当前已经创建的Migration Group，每个Group中可以包含一个或者多个LUN。本例中我们打算迁移一个已经存在的Group，“Internal ordering request system”，即内部下单系统所使用的LUN。单击这个Migration Group查看其详细信息，进入如图18-30所示的界面。可以看到这个Group只有一个LUN，而且这个LUN当前所处的存储介质Tier为High Performance – Mod Availability – RAID 5。LUN的容量为6.87GB，与方才创建的Tier中那个未被使用的LUN容量相同，所以可以迁移。

图18-29 Migration Group列表

图18-30 Migration Group详细信息

（9）单击Close按钮退出到图18-29所示的界面中，单击待迁移Group左侧的Migrate按钮，进入如图18-31所示的窗口。这里列出了所有可供迁移的目标Tier。其中有两个不能被选择，因为其剩余空间已经不能够容纳源LUN了。我们选择方才创建的Low Cost – SATA Drives的Tier，然后单击Next按钮，进入如图18-32所示的窗口。

图18-31 选择目标Tier

图18-32 选择源和目标LUN

（10）选择好源和目标LUN之后，单击Next按钮，进入如图18-33所示的窗口。此处可以选择立即开始执行迁移。还有一个Erase remaining data on source volumes的选项，如果勾选了这个选项，那么当系统迁移完毕之后，将向源LUN中覆盖写入0以消除源LUN中的数据以保证较高的信息安全等级。

图18-33 选择是否立即开始执行

（11）单击Next按钮进入如图18-34所示的窗口，为一个Summary窗

口，单击Confirm按钮之后，系统便会根据之前所设定的所有动作来执行这个任务了。迁移会自动在后台执行，整个迁移过程不会影响源LUN的数据访问。当LUN中的所有内容成功迁移到目标LUN之后，系统会做短暂的切换，切换之后，所有只能对源LUN的IO将会直接发送给目标LUN执行，之后系统根据策略可以删掉源LUN或者覆盖写入0以销毁原有数据。

图18-34 Summary窗口

（12）当迁移成功完成之后，再次单击对应的Migration Group，可以发现当前所处的Tier已经变为Low Cost – SATA Drives了，对应的LUN也是目标LUN，如图18-35所示。

图18-35 迁移完成之后的状态

6) 3PAR

Adaptive Optimisation为3PAR公司最近发布的数据分级管理模块，其实这个模块之前名为Dynamic Optimisation，即根据策略将Sub-LUN（Chunklet，3PAR的叫法）分布到不同的RAID Level、磁盘类型等存储介质中，已经具有了HSM的雏形。而更名为Adaptive Optimisation之后，变成了正统的HSM，增加了Tier的概念，形成了真正的层次。

3PAR存储产品已经支持SSD。Adaptive Optimisation为每个迁移对象提供3个Tier，每个Tier可以赋予不同的属性，比如磁盘类型、RAID类型、条带深度以及磁盘内外圈等。根据Sub-LUN（1GB大小）的IO热度（每GB数据每分钟的IO数量）以及其他用户制定的策略，系统可以针对每个Sub-LUN在策略的触发下在这3个Tier之间动态迁移。

7) Compellent

Data Progression是Compellent公司的数据分级迁移软件。Data Progression可以在存储端以Block为粒度进行分级迁移。至于Block具体为多大尚无从考证。下面对Compellent的分级管理进行简要演示。

(1) 如图18-36所示，Storage Profile是一种策略定义，每个Profile中会让用户定义3个Tier，这3个Tier性能由高到低。如果将某Profile映射给某个LUN，那么系统会自动将LUN中的数据根据访问热度向Profile中定义的高级别的Tier迁移。

图18-36 Profile列表

(2) 在Storage Profile上右击，从弹出的快捷菜单中选择Create Storage Profile命令，创建一个新的Profile，如图18-37所示。

图18-37 创建新Profile

(3) 如图18-38和图18-39所示，每个Profile包含3层Tier，Tier1的性能需要比Tier2高，Tier2需要比Tier3高。可以看到 Compellent在每个Tier的条件中只有一种可选，即RAID类型和磁盘数量。

图18-38 设定Profile中的3层Tier (1)

图18-39 设定Profile中的3层Tier (2)

(4) 选择了每层Tier的条件之后，出现如图18-40所示的对话框，为这个Profile起一个名字。

图18-40 为新建的Profile起名

(5) 名字填好之后单击OK按钮，进入如图18-41所示的界面中。

选中待迁移的LUN（图中Volume即LUN），右侧显示出了这个LUN的详细信息。

图18-41 待迁移的LUN详细信息

（6）如图18-42所示，在新建的Profile上右击，从弹出的快捷菜单中选择Apply to Volumes命令，出现如图18-43所示的对话框。

图18-42 将Profile赋予给LUN 图18-43 选择对应的LUN

（7）在图18-43所示的对话框中，选择需要对应这个Profile的LUN，单击OK按钮之后，系统便会自动根据Profile中所设定的3层Tier，根据策略及IO访问热度，在这3个Tier之间自动的动态地迁移数据了。

此外，Compellent还提供Fast Track技术，即系统不仅可以将数据放到对应介质的磁盘驱动器中，甚至还可以将数据放到对应驱动器中的不同磁道中（外圈、中圈、内圈）以实现更细粒度的分级，非常牛！

8) NetApp

NetApp对分级存储的理解似乎与其他厂商不尽相同。早些时间NetApp曾宣称分级存储将会被缓存管理所取代。NetApp并未实现传统意义的分级存储。NetApp的“替代”做法是在存储系统控制器内插一块或几块PCIE接口的内存卡，卡上插多条SDRAM，作为一个更大的Cache而存在，内部使用软件来将合适的数据预读入以便增加命中率。但是这个附加Cache不能用来作为写缓存，而且也不能永久保存数据，所以只能用来作为读缓存。

这种技术手段虽说可以增加读命中率，但是它终究还是作用在Cache层，并非作为一块硬盘存在，不能永久保存数据，不能作为写缓存，容量有限，局限性还是很大的。这种增加Cache的方法也并不能做到物尽其用，比如，访问频度高的数据可能依然被存放在低性能的SATA盘中而同时高性能的FC盘中尚有大量的剩余空间。靠增加Cache来提高命中率，就像是在一个内部并不平衡的系统中强行运化出阳气，久而久之必伤元气。比如，读命中率增加，读速度加快，那么主机客户端程序在处理完数据之后，需要写回保存的数据生成速度也就相应加快，而这个Cache并不能用于写，那么写数据就会积压在系统原本的Cache中，造成后端更加繁忙。也就是说，一开始就顾此失彼失去了平衡，那么后来也必然导致不平衡，从而大伤元气。然而笔者相信NetApp当前的做法只是一种不得已，将来或许也会开发真正的分级管理模块，但是至少目前来讲，NetApp在数据分级管理领域已经落后了。

总评： 目前看来，各个厂商所实现的分级迁移，远远没有达到理想目标，所能够设定的条件过少，自动化程度不够高，不够智能，提供开发接口的产品很少。要想达到ILM的最终目标，还有很长一段路要走。

7. HSM数据分级产品设计样例

本节介绍一个作者个人演绎出来的主机端HSM软件设计。国内存储软件行业在智能数据分级管理领域基本上无人问津，但是鉴于最近SSD市场逐渐趋于成熟，作者个人预测HSM将被像CDP、DR一样被火爆的炒作一把，其价值将会在几年之内完全榨取出来。在存储硬件方面，国内存储行业基本上没有掌握底层高端主流技术，但是在存储软件市场上，国内的厂商还是比较有作为的，虽然有不少也是在OEM，但是自

研的也有很多。所以，希望国产的存储硬件和软件早日脱颖而出占领市场。

SSD是近年来存储市场比较热门的主题，但是企业对SSD的兴趣好像一直不如个人浓烈，一方面由于成本的原因，企业如果选择使用SSD，则需要至少购买能做成一个RAID Group数量的SSD硬盘，这是一笔不少的开销；另一方面，由于SSD设计原因导致其寿命远低于机械硬盘，而企业环境下的数据IO很频繁，无疑会更快地让一块SSD寿终正寝；再一方面，即便真的使用上了SSD，要将哪些数据迁移到SSD上呢？用什么方法和标准来判断哪些数据需要放到SSD上呢？如果今天将某份数据迁移到了SSD，而过一段时间之后这份数据变得不再重要了，那么又得迁移走，换其他重要的数据迁移到SSD，这样维护成本是不是太高了呢？面对这一系列的问题，使得企业在选择和使用SSD的问题上犹豫不定。

有了这种市场需求，就可以针对需求，依附SSD和已有的ILM、HSM理论基础，来设计一款数据分级迁移管理软件。这个软件模块运行于主机端，在所有主机端的LUN或者文件系统之间做基于块或者文件级别的数据分级管理。现在的服务器一般自身都具有8个以上的SAS槽位，兼容SATA硬盘和SATA口SSD硬盘，所以，SSD硬盘只要插在主机本地即可，这样就不需要任何额外的投入。不管这台主机在使用何种类型的、何种厂商的、何种规模的外部存储设备，这些设备到了主机一层都作为LUN存在，而这个HSM软件模块就在这些外部LUN和本地SSD生成的LUN或者Space Pool之间做文章。比如根据IO热度将外部某个LUN中的某个区域迁移到SSD的空间内，当热度降低之后，再迁移回来。这个模块可以做到文件级和块级的迁移管理，底层使用文件层和卷层的过滤驱动设计。

另外，使用SSD做RAID，由于RAID的Parity分布很均匀，导致RAID组中每个硬盘在一个相当长的时间内接受的IO也是大致相等的，而SSD并非机械硬盘，所以其不会出现因为机械问题导致的故障。其主要故障原因就是在于写IO次数导致的寿命耗尽，如果一个RAID组内的所有SSD都差不多接受相同数量的IO，那么它们的寿命也会差不多，一旦某个时间段内某RAID组相继损坏两块或者多块SSD，那么其上数据就丢失了。

为了避免这个问题，在设计SSD的RAID算法时要引入一些额外考虑，需要将Parity不均匀地分布，要让某块SSD先坏掉，所以在这块盘上应放置更多的Parity。坏掉更换新盘之后，还要将其上多放置的Parity迁移到另一块SSD，因为如果新盘有更多的IO负载，那么其很有可能加速衰老，赶上其他盘，可能与其他盘同时坏掉，所以需要将额外的负载迁移到另外一块盘，而这块SSD会是下一块将要坏掉的，然后依此类推。主机端运行的这个HSM软件模块可以考虑这一点，实现软RAID而不使用主机端的RAID功能，也就是让主机端将本地插的SSD透传上来。

这个软件可以开发针对个人、企业、数据中心的版本。不同版本包含不同的功能模块。

说明： 以上仅为个人拙见，希望国产存储软件越做越强，赶超西方！

8. 判断你是否需要部署HSM

总的来讲，如果你公司的存储系统正在面临如下问题，那么恐怕你真的需要对数据进行分级了。

- 数据量庞大，存储设备众多。
- 性能分布不均衡，有些设备长期满负荷运行，性能低下，而有些则长期处于空闲状态。或者同一台设备上的LUN负载极度不均衡等情况。
- 有不同种类的存储介质，比如SATA、FC、SAS、磁带库等。

对于一个大企业来讲，数据总是不断增长的，而在应对数据增长的时候，相关人员不可能做到非常准确地预测，从而做出准确的采购计划，而这就使得企业IT存储系统内的资源不可能做到与企业的需求准确地对应，这样的话，就产生了浪费，或者性能分布不均衡的情况，导致生产成本增加。基于上面几个因素，可以利用一些监控工具来监控每台存储设备的利用率等情况，比如，调查一周工作日之内的统计结果，来判断系统总体的性能分布情况。

另外，你企业当前的存储系统架构也是一个重要的判断因素。你需要明白当前你的存储架构的现状，比如是否有很多信息孤岛？所有的主机是否共享一台或者少数几台存储设备？其次，你还要明确你对数据分级的最终期望，是想在少数几台主存储设备上实现分级，还是想在全局存储系统内实现分级？这些决定都会影响最终的部署效果。

如果既想要在全局范围内实现分级，而你的系统内不同厂商的设备过多，信息孤岛也过多的话，那么实现起来就是一件难度很大的工程了，需要大动干戈，伤筋动骨。所以，这就需要根据成本预算以及投入产出的比例来做综合判断。

9. 如何选择对应的**HSM**产品

对于企业而言，实现HSM可以有多种方法。比如你可以人为来判断

哪些数据是热点数据，从而将对应的文件手动迁移到高性能存储介质中；同样，你也可以将一些不需要的数据，用备份的形式来备份到磁带中永久存放。然而，对于前者来讲，你所能够操作的目标只能是文件，因为你只可以看到文件，不幸的是，在你手动迁移文件的时候，任何人都不能访问这些文件，同时，文件迁移完毕之后，你还需要将对应的目录路径指向新的存储位置。这一切都需要复杂的人为操作，而且操作之前必须制定计划以便最大程度地降低对应用系统的影响。而且，如果一段时间之后，你发现这些数据的热点期已过，需要再迁移回来，那么你就需要再次执行相反的动作。这无疑是个很大的挑战，对操作人员的技术水平要求非常高。

所以，需要一种自动化程度较高，而且智能化程度较高的HSM产品来协助企业完成数据分级。在选择一款HSM产品的时候，你可以根据上面所讲的HSM分类来选择，然后按照类别再来选择对应的产品。比如，你是想实现简单的文件自动迁移，还是想实现可定制复杂策略的块级迁移？是想在主机端实现迁移，还是存储端实现？是想在全局范围内实现还是某台设备上实现？

目前来讲，大多数厂商的数据分级产品几乎都是嵌入其自身的硬件的，比如EMC的FAST，只有使用了EMC对应的存储产品，才可以部署。3PAR、Compellent等公司的产品，也都是嵌入它们自身的阵列产品中的基于块级别的分级模块，也就是说，如果你的系统中没有这些产品，那么就不可能用它们的方案来部署数据分级系统了。

万幸的是，市场上的一些带有虚拟化功能的存储设备，比如HDS公司的USPV系列设备，它可以虚拟化后端大部分主流厂商的存储设备，然后在此基础上实现分级操作。但是这样的话，部署分级的同时还需要投入虚拟化这块，就有点买椟还珠的意思了，不过如果企业同时需要部

署这两种技术的时候，那么选择这种方案无疑是再好不过的了。

另外一种选择就是使用主机端的分级工具，比如IBM的Tivoli HSM，虽然它只是一个简单的文件迁移工具，但由于运行在主机端，所以无须考虑后端设备的多样性。但是所带来的局限性就是需要在所有需要分级的主机上都安装客户端。

对于系统中的NAS设备和SAN设备混合存在的情况，如果要在这两种设备之间做分级迁移，那么可供选择的唯一产品就是主机端的分级产品了。一些NAS虚拟化设备比如F5公司某款产品也可以做到在后端所有NAS设备之间做分级迁移，但是这只是针对NAS，而且分级策略也不是很智能、很详细。

总体来讲，企业如果要在现有的系统架构下嵌入式地部署HSM，那么难度是相当大的，而且可供选择的产品非常有限。但是如果企业想重新建立一套带有数据分级的新存储系统，而忽略原有的存储系统，那么可供选择的产品就一下子变得很多了，可以咨询前文所列出的这些厂商，他们一定会给你一个对应的合适的方案。

10. 存储厂商应该怎么做HSM

在上面的章节中我们可以看出，目前这些厂商的所谓HSM产品，其实并未真正做到HSM的核心层次，只是在表面上“意思”一下而已。对于IO属性的判断仅限于少数条件，有些甚至根本不提供自动迁移策略，完全靠用户手动将整个LUN迁移到目标Tier，这从根本上讲连HSM的边都不沾了。

如图18-3中所示的Service Level Objectives (SLO) 是一个非常复杂

而且难以形成标准的东西，业界目前尚未对STO有相应的标准。由于IO属性多种多样，不同IO属性之间可以相互组合，而且同一种应用在不同的规模、不同的时段都可以有不同的SLO需求，再加上前文中所说的政治和人为因素，所以这样看下来，SLO是一个具有颇多维度的东西。正因如此，存储厂商目前不可能做到HSM的理想状态，充其量多给出一些细节判断条件，比如Symantec的SF平台所集成的HSM模块那样。

要真正地做到按照SLO来分配Tier，首先要有具体的SLO，目前连SLO本身都还没有。所以，HSM下一步需要推进的就是定义SLO接口标准，用SLO作为连接应用层与存储层的桥梁，然后在应用层来实现HSM策略。迁移也由应用层根据策略来自动发起，或者由人自己发起，因为只有应用层和人才能够完全知道自己所产生的哪些数据需要何种性能或者特性的存储空间。

11. 从存储分级到存储系统全局资源分级/分配

信息生命周期管理和HSM目前被广泛认识为只作用在磁盘以下的层次中，对磁盘及其下层的存储介质进行分级管理，但是却忽略了磁盘以上的层次，比如RAM、CPU Cache。目前多数操作系统对RAM的管理都是大同小异的，而且几乎都是采用全局统一标准来管理，比如分配多少内存、多大的Page等。

然而，对于存储系统来说，它同时接受多个不同客户端的多种不同类型的IO流。到底应当按照什么样的条件来分配缓存资源呢？全局还是分区？Page回收到底应当按照怎样的条件来触发？这些都是需要仔细研究的。

提示： 以上这些都属于存储QOS（Quality of Service）的范

畴，诸如EMC、HDS等厂商都有对应的产品来实现QOS，具体可见后面的章节。

12. Tier和Cache之争

目前来讲，使用SSD作为一个Tier是大多数厂商都选用的方法，而有些厂商则使用SSD或者Flash介质作为一个大的缓存来使用。典型代表就是NetApp。

很早的时候NetApp就对SSD抱有疑虑，我记得当时其他厂商已经在着手开发动态数据分级了，而NetApp却犹豫的很，最终推出一块叫做Performance Acceleration Module (PAM)的PCI-E接口卡，专用于FAS3100系列。一开始其上是插DDR SDRAM内存条的，后来也有Flash颗粒版本的了。WAFL虽然是个很有特色的文件系统，但是其所存在的问题也是不可小视的，也就是经典的Sequential Read After Random Write的问题，即本来逻辑上连续的块，被WAFL处理之后底层却变得不连续了，这样在连续地址读IO的情况下，底层却表现为随机IO的行为，从而影响性能。PAM卡的推出可能也有这方面原因。其实所有文件系统多少都会有这种问题，只不过WAFL更加严重，而且WAFL将LUN也当做一个文件，这样的话对于本该比较刚性Block访问也变得左绕右绕，IO路径不等长，IO延迟变得很难预测。

那么究竟为何NetApp不使用SSD来解决性能问题，或者自己也开发自动分级存储模块呢？我猜测，这与其WAFL的原理有很大关系。SSD这东西，所有人看到它的表现，一定都是竖起大拇指的，但是我估摸着唯独NetApp对SSD具有那么一点点排斥心理，为何呢？

首先，SSD的出现，让WAFL的那一套写加速算法有点挂不住了，

包括全重定向写、尽力整条写等针对机械磁盘所作的大量优化，随着SSD的出现，一切都解决了，那么WAFL这一套势必在SSD面前就显得白费了，这一定让NetApp很难受的，其实NetApp一直都难受，即便是使用机械盘，WAFL依然面临着Sequential Read After Random Write（SRARW）问题，早就在研究新架构的WAFL了，比如是否可以支持RAID 5而不是RAID 4，是否可以不再重定向写了等等。但是对于WAFL这样一个复杂而庞大的架构来讲，牵一发会动全身，不是那么好改革的了。

第二，WAFL的重定向写措施，会迅速耗尽SSD上的剩余空间。懂点SSD的人都知道，SSD自己内部会去记录哪些page存有数据，哪些没有，这么做是为了损耗平衡算法，SSD内部也会有大量的重定向写操作，其做法与WAFL类似，但是WAFL这么做是为了方便地快照与整条写，SSD这么做纯粹是为了损耗平衡，不管怎么样，这两者是重复和部分冲突了。另外，WAFL不断地写到空余位置，那么SSD上的“曾经写过多少”这个高水位线就会迅速达到顶峰，SSD内部剩余空间迅速降低到最低值，严重影响SSD的性能，而WAFL的作用原理又不可能实时的将SSD中的“垃圾”块回收回来，因为WAFL从本质上讲可以认为是无时无刻不在产生垃圾（重定向写之后，以前的块便是空闲块了，但是SSD却无法感知文件系统层面的空闲块，依然认为是有用块），它根本来不及回收的，况且WAFL内部的两层FS之间已经为了忙活着回收空间而做了大量复杂流程了。如果说SSD让WAFL的优化变得价值全无，这一点还可以容忍，但是如果WAFL想用SSD而眼看着效果不好，那么就真的没治了。

第三，我们退一步讲，就算WAFL会很快耗尽SSD的剩余空间到最低值（也就是SSD厂商隐藏的那部分为了保证性能而预留的剩余空间，

比如100GB的SSD其实是有128GB物理空间的），效果再不好，但是也比机械硬盘要快，所以NetApp只能退而求其次将就着上SSD了。还有最重要的一点，别忘了，SSD目前的容量还太小，如果上SSD，会有两种用法，一种就是直接将SSD当做普通盘来用，做RAID，做Aggregate，做WAFL，然后做Volume，做LUN或者目录的Exports。但是这种做法适用的场景很少，比如一部分小容量的数据却要求极高的访问速度，那么没有问题，这种做法可以满足。但是如果遇到短尾型应用的数据访问场景，大量的数据却只有一部分为热点，那么此时你将所有数据都放到SSD上，显然是得不偿失，此时自然就需要有一种动态的细粒度的热点数据分级解决方案了，这也是目前几乎所有存储厂商都在搞的技术，而且主流厂商也都推出了各自的产品了。而我们回来看NetApp，它何尝不想推出自己的动态分级方案？它很难受，为什么呢？WAFL如果是老虎，那么NetApp可以说已经骑虎难下了。想在WAFL上引入动态分级子模块，不是那么容易的。动态分级子模块包含至少两个亚模块：一个是热点数据监控、统计模块；另一个是数据迁移模块。监控和统计子模块，可以作为一个旁路模块存在，不会对现有的任何FS架构产生太大影响，这个WAFL做起来没有问题，但是数据迁移模块，这对WAFL来讲，又很难受了。WAFL不按常理出牌，与其他传统FS不同，总是去重定向写，改一改就动全身，所以从技术上讲，实现动态分级还是太费劲，风险也很大，需要测试很长时间，所以我推测这也是NetApp迟迟没有推出动态分级的可能原因之一吧。

所以我估计，NetApp一开始就定下了基调，SSD目前来讲就作为大缓存的角色而存在。可以看到其新发布的FASx200系列，最高规格的FAS6280已经可以使用8TB FLASH的PAM卡了，当然，需要插多块PAM卡来堆叠成这么高的容量，而且两个控制器上的PAM卡规格必须对称，成双成对出现，而不能够只插一组卡让全局使用。

那么Tier和Cache这两种针对SSD的用法，到底哪个强哪个弱呢？我们来比较一下。缓存是实时预读，有很大的乱猜的成分；而Tier是长时间后台监测然后只迁移相对恒久热点，带有明显的目标，能发现长期的热点。所以Cache中的数据会随时迁入迁出，而Tier中的数据迁入迁出频率相对Cache要低得多，可以认为Cache是心急火燎，Tier则是慢工出细活。并且Tier是将数据直接迁移到SSD，但是SSD用作Cache的话目前厂商的做法一般是只支持对读IO数据进行缓存，写数据不缓存，这样就不能加速写了。

但是也不能一概而论，作为Cache使用也不一定非要使用传统的Cache算法，完全也可以使用更精细的热点监控和数据精细复制的算法，可能算法没有Tier那么考究，带有更加激进的性质。到底使用Tier还是Cache，得根据应用场景来综合判断，有时候并不能说谁优于谁。比如读多写少的环境，则可以选用Cache方式。最理想的一种解决办法就是，同时支持Cache或者Tier模式，而且可以随时触发或关闭Tier模式。

13. 数据分级管理之轮回论

我们都知道几乎所有操作系统对内存的管理，都使用Virtual Memory方式，用户看到的空间并不一定与物理空间一一对应，操作系统对物理内存实现了Thin Provision，比如明明只有2GB，程序却可以使用申请3GB的空间。为了弥补这个弥天大谎，操作系统只好使用拆东墙补西墙的方式，在硬盘上创建一个Swap分区或者Page文件，将这块空间也作为Virtual Memory的空间。当物理内存空间剩余到一定阈值时，操作系统将物理内存中不经常被访问的Page进行Page Out操作写入磁盘以腾出物理内存空间，而一旦程序需要访问一个处于Page文件中的Page，

那么操作系统执行Page In过程读入对应的Page到内存。操作系统的这种将不常被访问的数据迁移到Page文件中的做法，就是一种Tier分级操作。

纵观图18-5，我们可以发现，在CPU Cache、RAM、Disk这三者之间，操作系统的Memory Manager会负责将数据在三者之间做迁移，但是Disk自身以及Disk之下的所有层次，操作系统却没有提供任何模块在这些层次之间做数据迁移。所以，为了弥补这块缺失，HSM的催生是一个完全必然的结果。HSM并不是一个新东西，HSM其实是IBM用于在20世纪中晚期的IBM大型机上的一种技术，适配各种慢速存储介质。同时代或者更晚的一些主机系统比如Alpha/VMS也都在使用这种技术。

其实对于外部存储系统的Thin Provision+Tier，其本质与操作系统对内存的管理方式完完全全是一回事。其实计算机系统内的很多技术，都表现为一种轮回和嵌套，这种例子在本书其他章节还有更多。这种轮回和嵌套，看似是人类智慧使然，其实是世界之本质，也就是阴阳使然，阴阳变化的过程就是一个轮回的过程，分分合合，而阴阳不断的叠加过程就是一个嵌套过程，一层层的底层逻辑组成更高层的逻辑，而这些逻辑之间的共性，就是阴阳轮回嵌套。而在发明新技术的过程中，先想一想是否可以基于已经存在的技术的设计思想来做一个模拟，可能就会大大降低设计成本。笔者之前在做某项目的时候，曾经遇到过这样一件事：某程序员为了解决某个问题，设计了一套代码，非常便捷地解决了问题。但是最后另一同事在读代码的时候却发现，咦，这不是Windows MFC下的某个代码模块么？原来该程序员自己写出了一个MFC早已写好的封装代码。

14. 星星之火可以燎原——火星高科MSP数据分级中间件平台

上文曾经提到过，目前厂商的HSM产品基本上只考虑了后端性能方面，而基本忽略了人为因素以及其他更多的SLO，造成迁移引擎在进行所谓“智能”迁移的过程中只有一个指挥棒，那就是访问频度。这一点其实也无可厚非，从某种角度来讲，在一个SSD—FC/SAS—SATA的层级组中，不管数据处于何种层级，它们的访问速度差别并不像磁盘到磁带这种差距，加上磁盘上层路径各处的缓存，到了应用层之后，这种差距会更小，尤其是不具累积效应的偶尔访问的时候，虽然可能会慢一两秒。但是这种速度降低是可以被接受的。所以从这种角度来讲，存储端当前的HSM产品还是有很大意义的。

但是对于归档领域，或者一个容纳了几乎所有存储介质层级并在全局范围内做HSM管理的系统来讲，如果只按照某个单一条件来触发迁移，那么就会产生与人为因素的冲突。这种情况下，要么就在这个系统中引入更加复杂的条件，引入大量的SLO，要么就干脆把HSM中的上层策略部分抛掉，只留下一个底层平台，然后提供一些控制接口给上层的策略引擎。也就是在图18-3中所示的HSM Policy Engine与HSM DataMove Engine Switch之间形成一个公开的接口层，将整个HSM系统在此分割为上下两部分。应用系统厂商可以对上面的部分加以开发，形成自己的迁移策略引擎模块，而存储厂商则专注于下层的开发，根据需求不断完善和丰富接口。鉴于存储厂商众多，一时间不可能统一接口，所以自然也就出现了中间件平台。

火星观点： 存储端不适合做具体的迁移或者分级策略，只需要提供迁移接口。策略和触发条件要由应用层或者人来制定。存储端要充当一杆枪而不是使用枪的人。

火星高科（北京亚细亚智业科技有限公司）的数据迁移管理中间件平台Mars Storage Platform（MSP）就是专门为数据迁移和归档管理所设

计的一款中间件平台。作为一个迁移/归档中间件，首要的就是要兼容各种主流的磁带库、自动加载机、磁带驱动器以及VTL，不但如此，还要支持和融入各种SAN和NAS环境，甚至一些并行文件系统环境。然后，最重要的一点，作为中间件，需要有丰富的查询和控制接口。以上的条件MSP都做到了，MSP支持数百种磁带设备；支持在SAN和NAS环境中部署；可融入各种SAN共享文件系统；提供了30多个API函数，包括归档和回迁管理、任务管理、对象管理、磁带管理、带库管理、存储位置管理、磁带复制、磁带池管理等多个功能类别。

能够提供二次开发接口的归档平台产品非常少，国外产品中IBM Tivoli与Symantec等可以提供接口，但是Tivoli的部署复杂度、易用度、先期部署成本、后期维护成本都是高不可测的。目前国内已经有多个厂商的不同的数字媒体信息系统在后台使用了MSP中间件来作为归档和迁移。

1) MSP的逻辑架构

如图18-44所示，整个MSP由13个模块组成。其中服务器模块负责接收操作指令（通过API或者用户手动发起）并控制迁移器对数据做实际的迁移；收发器负责处理对象（见下文）；带库控制器则类似于驱动器、机械臂的驱动和管理层，向带库发起IO请求必须经由这个模块；冗余模块是指专门用来做磁带RAID技术的模块（见下文）；单机读取模块是为了解决在回迁数据的过程中由于已经出库的磁带距离中心带库较远而不方便入库的问题而生的，可以直接在出库磁带所在的地点假设一台服务器以及磁带机，其上安装单机读取模块，这样直接就可以从已经出库的磁带中将数据读出从而回迁；开发接口模块便是MSP所提供的查询/操控API了。其他模块不再介绍，按字面意思理解即可。

图18-44 MSP架构图

2) MSP的部署形态

如图18-45所示，MSP中的各个模块可以分开安装于不同的服务器上，当然也可以将它们都安装在一台服务器上。其中迁移器、收发器这两个模块必须连接并且可以看到后端的存储空间，因为这二者需要对后端存储空间中实际的数据做读出/写入的迁移操作。带库控制器模块必须能够连接到带库设备。

图18-45 MSP部署形态

用户可以通过系统管理中心来定制各种策略以及归档和回迁任务。如图18-46所示为系统管理中心的界面。

图18-46 MSP配置界面

3) MSP的优点

- 支持多级存储，比如磁盘、VTL、磁带、NAS网络路径。
- 支持多个迁移器集群化并行以提高速度和吞吐量。
- 面向对象的迁移粒度，可以将多个互相有逻辑关联的文件组成一个组对象，并以对象为单位进行迁移。

火星技术： 比如某监控点，在多个角度都安装了摄像头，各产生一路视频流和视频文件。此时就需要将针对这个监控点的所有角度的视频文件存放到一起，而不是分开无序存放，这样，在视频回放的时候，就可以一下子选出这个点的所有角度的视频来观看，而不是从多个存储位置或者磁带各自将对应的视频抽出。这种方式在MSP中被称为对象存储模式。

- 具有磁盘缓冲技术，可以将磁带中的部分访问频度较高的文件复制到磁盘中作为Cache缓存以提高访问速度。

火星技术： 用磁盘作为磁带的缓存就像用RAM作为磁盘的缓存一样，MSP将这种思想用到了磁盘下面的层次。注意，这里的缓存并不是指D2D2T那种模式。缓存和缓冲不一样，缓存指的是慢速介质上的数据被复制到高速介质中，高速介质作为一种Cache；而缓冲则表示数据先存放在高速介质中，作为一个Buffer，此时低速介质中尚未存放这些数据，待高速介质达到一定触发条件后，数据便被移动或者复制到低速介质。

- 使用通用标准协议来控制主流厂商的带库，这就使得MSP兼容几乎所有带库。
- RARM（Redundant Array of Removable Media）磁带冗余技术，有效防止因磁带数据损坏而造成的数据丢失。

火星技术： RAID的意思是“廉价冗余磁盘阵列”，磁盘可以阵列，磁带一样可以。火星科技的磁带冗余技术，就是将磁盘阵列中的RAID技术迁移到了磁带中。同样，用多盘磁带来做XOR运算，生成一份校验磁带。当某盘磁带完全损坏，或者其中部分数据损坏时，可以用XOR算法生成损坏的数据。不仅如此，如果多盘磁带上都有部分数据损坏，只要同一条带上最多只有一个segment的数据损坏，那么一样可以通过校验来找回损坏的数据。

- 迁移器带宽控制，可以针对每个迁移器或者全局来限制迁移数据时的IO强度，使得迁移操作对生产环境的影响降为最低。
- 文件片段恢复技术，可以从某个特定格式的大文件中恢复出可以

被应用程序所识别并且使用的文件片段而不是回迁整个文件。

火星技术： 比如在视频回放环境下，有时候用户可能只需要查看某个大视频文件中的某一小部分，而此时如果大动干戈地把整个文件都回迁，不仅浪费时间，而且还有损设备的寿命，还占用额外的磁盘空间。对此，火星科技开发出了文件片段恢复技术，通过对特定格式的文件做特殊标记，可以以很小的粒度来恢复出应用程序可以识别并且正确播放的视频片段。

- 开放式的磁带记录格式，第三方软件也可以识别。
- 支持多种主流OS平台，包括Windows、Linux、Solaris、HPUX、AIX。

4) 在用户苛刻的需求中成长起来

火星高科在技术领域有自己独特的观点，这些观点都是他们在面临了很多实际客户环境并且参与定制开发后逐渐总结而成的，具有很高的参考价值。

火星事件： 火星MSP平台以及企业级备份软件均支持磁带离线存放，即Vault功能。这样就可以使用较少的磁带槽位来满足对大量磁带的操作。某用户的离线磁带架特别多，每次软件提示将某某磁带放入带库的时候，用户查找起来很麻烦，用户将磁带架进行了拍照，并且将照片以及对应的架子以及槽位编号发给了火星科技，要求将图片做到软件中，每次对磁带进行出库或者入库，均在图片对应的位置进行直观的标识。这还没完，没过多长时间，用户的磁带架升级了，架子上方增加了LED显示屏来显示对应的信息，用户提出要求，让软件直接可

以通过RS—232串口及对应协议直接在LED显示屏上显示出磁带信息。

通过一次次对用户各种需求的定制开发，火星高科快速积累了大量经验，将很多有共性的定制功能放入了软件主体中，不断丰富软件的功能。

火星事件： 高级分级应当手动进行，越高级的用户，他想手动控制存储系统以及数据的欲望就越强烈。美国的分级产品之所以自动化、智能化强，是因为美国的人力成本高昂，而中国人则比较勤劳。

某档案馆的归档项目中，火星高科遭遇了大量小文件备份速度奇慢的问题，这个问题是普遍存在的，而火星高科没有回避，选择了冲破这道堵了很长时间的墙，取得了胜利。

火星技术： 大量小文件的传统磁带备份一直是让人非常头痛的事情。文件索引及Catalog的生成和处理过程是很慢的，大量小文件的情况下，可能光这一步就要耗费数小时甚至几天的时间，这显然无法接受。对此，火星高科技巧妙地避开了索引过程而转为利用一种变通和虚拟的方式来备份大量小文件，速度获得了20余倍的提升。

某用户的数据备份项目中，竞争对手为国外某备份软件厂商，他们的销售一个劲地向用户忽悠他们的多流备份技术，即同一个备份任务拆分成多个数据流同时写入多个驱动器，由于火星高科的备份软件不支持这种方式，所以对手一直拿这一点来攻击。而火星高科从容应对，成功赢得了用户的信任。

火星观点： LTO已经发展到了第4代，第5代也已经出来了，

数据传输速度已经非常高了，待备份的应用程序向外吐数据的速度甚至都可能达不到这个速度，既然这样，再将它拆分成多个流，有必要么？不但没有必要，反而有害，第一，多流备份之后的磁带一旦有一盘损坏，那么其他磁带上的数据也就没用了；第二，本来前端的数据速度可能已经连一个流都饱和不了了，再将其拆分为多个流，那么势必造成磁带驱动器欠载，造成鞋擦效应（见其他章节），降低磁带寿命。

多流备份是早期的技术，那时驱动器的数据写入速度很低，所以不得已而为之，但是随着硬件的发展，自己、这个问题早就被弱化了对手厂商拿着鸡毛当令箭的做法，实属不该！

关于备份出错是否要自动重试的问题，火星高科也有自己独特的看法。

火星观点： 备份出错大多数时间都是因为带库硬件或者磁带等方面出问题，此时就算重试N次也无济于事，反而还浪费了系统资源。硬件原因导致的备份出错无须盲目重试，待手动排除故障之后，手动再次发起备份。

火星高科的很多观点和技术都颇具中国特色，这正是因为火星高科是从中国各行业用户的苛刻的、千奇百怪的需求当中成长起来的。

火星高科成立于1992年，至今已经快20年的时间了。2010年中旬笔者有幸参观了火星高科，并且与同仁们进行了热烈的交谈。从交谈中笔者感觉到，火星高科是一家非常重视技术的公司，以用户需求为导向，技术研发为依托，如今公司已经成为一家国内数一数二的在归档领域掌握自主核心技术的公司。从当初对磁带一无所知到今天成为归档领域的权威，火星高科正如她的名字一样，通过不断的努力进取，星星之火终成燎原之势！

畅想：作者在此也以个人的角度畅想一下MSP的未来。下一步MSP可以向全面数据分级中间件方向进军，内部定义各种Tier，丰富各种接口，比如系统内的Tier分类和查询、Tier的性能属性查询等。功能上，除了支持文件级归档，还可以与底层存储设备配合，支持块级或者LUN级的透明分级，并且提供一系列针对块级别的API，比如将某个LUN从RAID 10的RAID组迁移到RAID 5的RAID组，或者将某个LUN中的数据迁移到NAS上，并且还保持客户端访问方式不变。当然，要做到这些功能是很不容易的，这要求这个平台既具有迁移中间件的性质，同时还要具有虚拟化网关的性质。

5) MSA备份一体化设备

火星高科最近发布了一款备份一体化设备，叫做火星舱（Mars Storage Appliance, MSA）。火星舱数据备份设备支持多种操作系统平台，如Windows、Linux、IBM-AIX、SUN-Solaris、HP-UX、RedHat、Novell、红旗等，支持这些异构平台下的文件、Oracle、SQL Server、Sybase、Exchange等数据库备份以及操作系统备份，支持重复数据删除功能。同时允许系统管理员将硬盘分级，将特定的硬盘/分区完全仿真成磁带库或者磁带机，也就是VTL功能，可大幅度缩短备份/恢复时间，同时减少高峰期对网络资源的占用。并可不限驱动器数量，提供用户分类存储数据功能。如图18-47所示为火星舱系列存储设备中的MSA Backup Advanced（MSA-BA）一体化备份设备的系统架构图。

图18-47 MSA-BA一体化备份设备

其中备份策略引擎负责全局配置、管理和触发整个备份域中的备份任务，数据移动器负责将数据从各个待备份主机端相应的应用代理处传

送到备份服务器中的虚拟磁带中保存，设备控制器是专门用来控制虚拟机械手的模块，VTL模块则是用来将底层磁盘虚拟成对应的虚拟磁带驱动器、虚拟机械手以及虚拟磁盘的虚拟化引擎模块。

6) 支持国产，并不应该只是口号

近年来，国家有关部委也明确了态度，在涉及国家安全级别的项目中优先或者必须使用国产产品。并且于2010年5月对信息安全产品要求必须通过CCC强制认证，包括备份软件。但是现状是，没有通过CCC认证的产品依然存在于市面上。火星高科的Mars Backup Advanced企业级备份软件支持主流数据库的备份，包括Oracle、Sybase、SQL Server、Exchange和多家国产数据库产品，如人大金仓、武汉达梦、神州奥斯卡、TRS等，也于第一时间通过了CCC认证。对信息安全产品的CCC认证过程中包含很多项目，比如对后门、漏洞等的审查等。然而对于一些国外的产品，未通过3C认证的也还在卖。政策监管和执行的不到位，让政策成了一纸空文。

无语： 不仅如此，界定一款产品是否是国货的底限，也被打破了。在最近某个数据备份项目中，招标明确规定：投标产品必须是国产产品。但是结果却让人瞠目结舌，中标的竟然是IBM，它的产品竟然被定义成了国货，这个结果让参加投标的多家国内厂商感到无语。不知道“有关部门”到底在干什么，强烈建议有关部门把“有关部门”取缔。

在此也号召国内广大用户，能用国产尽量用国产，不仅考虑价格因素，至少要考虑安全、民族、国家。多一个人用国产，国货的品质就会提升越快，就会早日赶超西方。

18.3.4 Deduplication（重复数据删除）

所有人的电脑内总会有一些重复的文件复制，尤其是对一些有收藏癖好的人来讲，他们会疯狂地在Internet上下载一些他们认为很有价值的视频、音乐或者图片等，由于他们在不停地下载而很少去欣赏这些内容，所以遇到相同的内容时可能会无意识地再次下载，这样的话，他们硬盘上的数据将会有相当数量的重复。这些重复的数据存在于存储系统中，一是浪费空间，二是一旦需要备份，那么又会浪费备份时间，而且备份之后的数据也会再次占用额外的空间。

在企业IT系统内，包含了个人电脑本地存储、服务器本地存储和外部网络存储系统。在这个整体的系统内，数据的重复率将会更大。比如对某个数据库的备份，每周一次全备，那么两次全备份的数据一定会有很多相同的部分。

再比如，每个员工电脑上至少都会装一套Windows操作系统，这些数据在全球范围内，就是重复的。而对于服务器，除了操作系统重复之外，安装的程序也有可能重复，而程序所生成的数据也有可能重复，比如某人给企业内所有人发送了一封邮件，并且携带了一个10MB大小的附件，那么邮件服务器会在每个人的邮箱中都生成一份这个附件的复制，这就是一笔很大的重复数据。

说道邮箱，腾讯QQ邮箱是笔者一直以来比较喜欢用的。QQ邮箱对超大附件的做法就是将附件存储在服务器上一个独立位置，不管将其作为附件发给谁，这些附件始终只有一份复制，任何收到邮件的人所看到的附件其实并没有附到本封邮件中，而只是一个链接，打开附件时会从服务端将数据下载到本地，这是一个很好的解决办法。那么对于处于不同机器之上的本地文件如果重复，能有什么办法来解决呢？在此总结一下所有可能的场景。

(1) 同一主机操作系统本地存储的重复文件。包括同名但是内容不同、内容相同但是名字不同以及只有部分内容相同的文件（下同）。

(2) 不同主机操作系统本地存储的重复文件。

(3) 不同主机操作系统都在同一台外部NAS存储设备进行数据集中存储时的重复文件。

(4) 不同主机操作系统都在同一台外部Block存储设备进行数据集中存储时的重复文件。

(5) 虚拟机环境下磁盘镜像文件中的重复部分。

(6) 虚拟机环境下裸磁盘映射模式下的重复数据。

对于(1)，可以由用户手动来判断并且删除重复的文件，必要时可以对一份文件作不同名称的快捷方式或者链接。但是对于只有部分内容相同的文件，如果不通过特殊程序的话，就只能保留。

对于(2)，在这种情况下，多台主机上的操作系统文件大部分是重复的，为了消除这种重复，可以部署无盘系统，但是无盘系统的性能实在不好控制，而且使用起来有诸多不便。而对于非操作系统文件的重复文件，是无能为力的。

对于(3)，NAS系统的不同或者相同目录中有重复内容的文件，这种情况下如果没有特殊的程序起作用的话，不能够擅自删除文件或者制作链接。因为NAS一般只提供对外的接口，其内部虽然也是某种操作系统，但是NAS设备一般不会提供用户其内部文件系统的操作接口，所以这种情况下的重复文件也是无法消除的。

对于（4），由于重复的文件存在于不同的主机操作系统内，那么如果要删除掉重复的文件，就必须将最后留存的那份实体文件所在的目录通过网络共享出来，其他主机访问这个共享，并且制作一个针对这个文件的链接，那么这个链接就只能是一个快捷方式，快捷方式使用起来有诸多不便。所以，这种情况下要实现重复数据的删除也是不现实的。

对于（5）和（6），就更不是通过用户手动能够实现的了。

综上所述，在同一主机系统内，对于整个文件的内容重复，可以在最终用户层消除重复，但是需要用户来手动操作并且记录链接关系（即便是同一主机操作系统内，硬链接不能够跨不同文件系统），这显然不现实。对在不同主机系统内的重复文件，无能为力。所以，不管从任何角度来讲，都需要一个特殊的程序来实现这种重复数据删除工作，并且还需要维护对上层访问的完全透明性。

Single Instance Storage（SIS，单一实例存储），就是实现这种删除重复文件内容的一种技术。所谓Single Instance（单一实例），指的就是相同内容的文件，在系统内只存在一份实体，其他副本都只作为一个指针链接而存在，链接只占用一个FS Block的空间。SIS可以在主机端实现，比如在某个操作系统上安装一个SIS处理模块，它会根据用户的设置来自动定期或者实时地扫描系统内所有可访问的文件系统内的文件内容，一旦发现相同内容的文件，则只保留一份文件实体，删掉其他多余的副本并为它们创建特殊的链接（可以跨文件系统，并非普通硬链接），这个链接其实与上一节中所述的Tier分级软件所做的链接类似。

对于上文中（2）对应的情况，虽然SIS理论上当然可以实现跨主机的重复数据删除，但是由于需要跨网络访问，在管理和性能上皆有诸多不便，所以目前尚未有这种实现方式。也可以在NAS上实现SIS，这样

就解决了上面的（3）所对应的问题。而对于（4）中的情况，其实是与（2）类似的，只不过数据放在外部块级存储设备而不是本地，所以SIS在这种情况下也不能实现重复数据删除。对于（5）所述的情况，SIS是无能为力的，因为即便是用相同的步骤安装两台GuestOS，这两个OS的系统盘对应的磁盘镜像文件也不见得每个字节都相同，虽然它们很大一部分都是相同的。对于（6），SIS此时已经看不到文件了，所以更是无能为力。（对于（5）和（6），虽然无法在虚拟机服务器上使用SIS达到预期效果，但是依然可以在GuestOS中使用SIS技术来消除重复文件。）

SIS可以理解为文件级别的Deduplication。块级别的Deduplication才是真正意义上的Deduplication，简称Dedup（去重/消重）。但是一般情况下会用Deduplication同时表示文件级和块级的重复数据删除。文件级的Dedup不但在满足实际需求上存在上文所述的一系列问题，而且其底层技术实现方法上也有很大的局限性，比如，只能以整个文件的二进制内容来比对，而遇到只有部分内容相同的文件，或者只有文件头部信息不同而后续的内容完全相同的文件，则无能为力。

但是不排除有一些稍微智能一些的SIS技术，可以识别和定界并且保留文件头部信息。比如mp3等音视频，mp3文件的头部会保存一些音乐信息，如果有两个mp3文件，其实体音乐编码部分是完全相同的，只不过歌手、类别等头部信息不同，所以这两个mp3文件的大小也不同，那么，对于高智能SIS便可以识别这种头部信息的不同而将其保留，然后将重复的音乐编码部分删除。然而，要实现这种智能，就必须感知各种主流的应用程序所生成的文件比如MS Office、mp3、mp4等，并且随着应用程序的升级换代，SIS模块也需要跟着更改才能识别新格式的头部信息和边界，而且应用程序有太多，不可能每种都支持，所以实现起

来太累。需要有一种更彻底的、更高效的、一劳永逸的解决办法。

这种一劳永逸的Dedup方法就是块级Dedup。Block是存储系统路径中仅次于Sector的最底层的数据结构了，如果直接来比对整个存储系统内所有Block的二进制内容的异同，消除相同内容的多余Block副本，那么不管Block中存储的内容对应的是哪个分区的哪个文件系统下的哪个文件，相同内容的多余Block的实体内容都可以被消除，而只在Metadata中留有一个指针来指向被保留的那唯一一份有实体内容的Block。

如果多台主机将自己的数据存储于同一台外部独立存储设备中，如果在这台存储设备上实现全局的Block级的Dedup，那么也就可以做到消除多台主机上的重复数据。对于虚拟机服务器上的磁盘镜像文件，块级别的Dedup就可消除两份镜像文件中冗余的Block，而对于利用裸盘映射模式来存储GuestOS数据的虚拟方式，同样也可以使用块级Dedup来消除重复数据。所以说，块级的Dedup是最彻底也是去重比率最高的Dedup方式。现在基于SIS技术的产品不多见了，基本上都是基于Block级的Dedup产品，相对于智能SIS，Block Dedup设计成本更低，去重比率也更高，获得的收益也更大。

以上介绍了Dedup的必要性，以及实现Dedup的两种大方向。下面就对实现Block Dedup的具体技术进行介绍。

1. 压缩与Dedup的本质区别

想到节约空间，大家可能首先想到的是压缩。比如WinZip、WinRAR、7-Zip等工具，都是我们常用的，而且某些情况下会获得很高的压缩比。但是有印象的读者可能还记得，本书第16章中16.2.5节中的那个例子，普通压缩工具对那种情况是无能为力的，因为压缩是一个局

部处理，而并不是全局处理，窗口很小。

而Dedup所要实现的，是针对整个存储系统的全局数据，普通压缩程序不可能去扫描全局的数据然后压缩而后访问的时候再解压缩，这是绝对不现实的，一是耗费大量CPU资源，二是对上层很难做到透明访问（WinXP内置的Zip引擎可以做到透明访问）。所以，需要使用其他技术手段来实现全局Block Dedup。

2. 全局范围内实现Dedup的核心技术手段——Hash

每个人的指纹都不相同，一旦发现两个相同的指纹，就证明是同一人所留下的。如果给每个Block录一个指纹并保存，那么就可以通过比对这个指纹来判断两个Block是否内容相同了。对数据来做指纹录入，有一个现成的方法，即Hash方法，Hash方法衍生出多种不同的具体算法，目前最为常用的算法是MD5（Message Digest v5）和SHA-1（Secure Hash Algorithm v1）。SHA-1算法会对任意一份长度小于264的数据内容进行扫描计算最后得出一个长度为160的值（MD5算法则是生成一个128的值），这个值就是针对这份数据的指纹。相同内容的数据经过Hash之后，总会得到相同的指纹，但是不同内容的数据在经过Hash之后，也有一个非常小的几率可能会得到相同的指纹。

从实体内容可以算出指纹，但是却绝对不可能从指纹逆算出原来的实体内容。所以Hash方法也常被用于密码保存和比对，即系统不保存明文密码，而是在用户设定登录密码时将密码Hash成指纹存放，这样，即便黑客从系统中截取了这段指纹，那么也不可能根据指纹逆算出原来的明文密码，即其他人永远不可能知道是哪些字符串被算成了对应的Hash值。用户登录时输入的密码也被计算成Hash值与系统保存的指纹进行比对，如果匹配则成功登录。破解Windows登录密码的过程其实并不

是“破解”，而是暴力地将系统中保存Hash值的文件替换成其他文件，新文件中的Hash值对应的原文密码是已知的，利用这个方法，用已知的密码即可登录系统。

不管数据的长度有多长，只要不超过额定大小，用同一种算法所计算出来的指纹长度总是定长的，比如即便针对一个大小为4GB的文件用SHA-1算法来提取其指纹，指纹长度依然为160。那么如果某个系统内有两个4GB的内容重复的文件，那么它们的指纹也一定是相同的，此时就可以删掉其中一个文件的实体内容而只保留一个占位符，当有程序访问这个被消除实体内容的文件的时候，Dedup模块会根据指针信息从剩余的那个文件中对应的部分将内容提取并且返回给访问者。这样做，相比压缩来讲就有了本质的变化。

提示： 数据指纹技术有很多神奇的应用，比如某网站提供一种服务，如果你会哼唱某段曲子但苦耐不知道这首曲子的名称，那么只要你哼哼出这首曲子并且录下来上传到服务器，那么服务器经过短暂分析之后便会告诉你这首曲子的名称。乍一听这是个非常神奇的功能，其实如果明白数据指纹技术，就不会诧异了。服务器会根据这首乐曲的音调频率等信息生成对应的指纹，然后与数据库中的大量已经保存的音乐指纹比对，如果发现与某个指纹类似，那么便将匹配的音乐名称返回给请求者。当然，这种音频指纹比对算法需要具有一定的Robust性，会过滤掉环境杂波的编码数据，Hash是做不到的，即使音调音色都相同的两份音频，其编码之后的数据也不一定相同，所以只扫描计算二进制流的Hash方法是无法完成比对音频的任务的。

数据指纹的另外的应用领域还在于判断一份数据在传输的过程

中是否发生了内容改变，在传输的源端算好一个指纹，目的端接收到之后再算一个指纹，比对两个指纹，相同则证明没发生改变，不同则表明发生了改变，数据不可用。

另外，数据指纹还可以用于远程Cache环境。比如某个CDN内容发布网络环境中，针对一份源数据，在各地拥有多个二级Cache，这些Cache需要严格保证其Cache的数据与源数据内容一致，如果源数据发生内容改变，那么其指纹也就改变。

Cache可以从源端拉取指纹随时比对，如果发现不匹配，则表明源已经改变，那么Cache的数据不再可用，针对用户的数据请求，Cache会从源端把最新的数据拉过来。

数字签名。每个人的笔迹都是不同的，但是e时代的电子签名如何表示呢？答案是使用Hash算法来将某人给出的一串特定字符算成指纹，只要给一份电子合同附属上这个指纹便知道某人在合同上按了手印。其他人不可能伪造别人的签名，因为其他人根本不知道别人的原始字符串。

其实Hash算法在各种程序中都被广泛使用，比如OS内部、数据库类程序等。只要涉及到用很少的数据来唯一表示一个很长的数据，那么都会用到Hash算法。

网络硬盘服务商可以使用Hash技术来实现单一实例存储。比如腾讯的QQ中转站就使用了Hash技术，每个人在上传大文件的时候，系统首先会对文件进行“扫描”，这个扫描的过程其实就是计算Hash值的过程。浏览器将Hash值传送到服务端的某数据库中进行匹配查询，如果发现了相同的Hash值，则表示已经有其他人上传了相同内容和文件名的文件，那么本次上传将会立即完成，不管文件有多大。

3. Hash冲突及其解决办法

没人能够保证地球上所有人的指纹都是不同的，很有可能某两个人的某处纹理是可以吻合的。数据指纹也一样存在这个问题，算法决定了针对不同的数据内容有可能生成相同的指纹，但是几率非常低。这种情况称为Hash冲突。设想一下，如果有两份明文内容，它们的指纹却恰好相同，而且其中一份内容恰好是某人的密码，而你恰好知道这件事，那么你此时虽然不可能知道那人的密码明文是什么，但是你却知道用你的明文可以算出和他一样的Hash值，那么你就可以用你的明文去登录他的系统，取得他的权限。

然而，寻找Hash冲突的字符串，或者根据已知Hash值来寻找拥有相同Hash值的字符串，这个过程虽然是大海捞针，但是理论上还是可以捞的。我国山东大学王小云教授早在本世纪初就宣布她已经找到了针对已知的Hash指纹来找到与已知Hash值相同指纹的数据的方法。请注意，这种方法并不是Hash的逆运算，因为Hash是不可逆运算的。我们设计一个简单证反即可得到这个结论：假设可以根据Hash值来逆运算出原始值，假设有这样一种方法，那么逆运算出来的原始内容只可能有一个而不可能有两个结果，那么这个结论就与“不同原始内容可能会有相同的Hash值”这个事实相矛盾，由此就推翻了这个悖论。也就是说，原始内容M1其指纹为H1，通过某种算法，可以找到另一份原始内容M2，而其指纹也为H1，但是无法判断M2是否等于M1，可能等于也可能不等于。但是是否等于已经不重要了，这就像DNA不同但是长相相同的两个人一样，他们在社会中某些场合可以互相替代，但是一旦你和他说话，就可能会发现根本不是一个人。

计算机领域与社会不同，高层智能尚未在计算机世界出现。只要有了Hash值这个通行证，就可以伪造电子签名，比如某员工知道老板的电子签名Hash值，而后他根据这个Hash值找到了某个Hash相同的另一份字

符串，那么他便可以用这份字符串来对某份伪造合同进行签名，而系统对这份字符串的Hash计算结果与老板的相同，所以系统认为合同是经过老板签名的。这种方法虽然并不是真正意义上的“破解”，但是其造成的后果与破解无异。其被称为“Collision Attack”。

Dedup系统中如果出现了Hash冲突，如果不想办法解决，那么对应的数据块就会永久丢失，丢失还不是最严重的，最严重的是对业务产生致命影响：本来两份数据不相同，但是由于指纹相同，系统只保留了一份数据，当业务层要访问被去重的那份数据时，系统返回的却是与之前不同的数据。而面对这种情况，业务层一般会感知到所得到的数据格式根本不是自己想要的，所以会提示出错。但是如果刚才的那两份数据恰好具有一定的相似性，或者数据为完全的裸数据，其中不包含任何应用可识别的高层格式，那么数据的变化就不会被业务层感知，从而在错误的数据之上继续进行处理，那么可能会得出业务层的重大决策失误或者重大生产失误。

要完全杜绝Hash冲突的唯一办法就是在当比对两个Hash值之后发现二者匹配，那么可以在此基础上再对原文内容进行二进制的逐位比对，或者取两份文件的数个相同部位，比如头、身、尾来做二进制比较。但是，如果在重复数据相当多的情况下，这种做法非常耗费计算和IO资源。由于Hash冲突的几率本来已经非常低，如果想办法把冲突几率降得更低，也是一个可行的方法，基于此，可以同时使用两种Hash算法，比如MD5和SHA-1来针对目标数据进行计算得到两组结果，然后分别比对每一组结果，如果两次比对都相同，那么就可以被判定为重复数据。这样做虽然也不能完全杜绝冲突，但是其几率又被大大降低了。

4. 如何设计Dedup

如果换了是各位，如何基于Hash技术来设计一个Dedup系统呢？下面笔者就从简单演绎到复杂从而来纵观Dedup的全貌。

设想有一台Windows服务器，挂载了外部存储系统中的几个LUN，都格式化成NTFS文件系统，这些文件系统下存有大量的冗余内容，现在需要设计一个Dedup模块，在保持对文件访问的透明性的前提下消除重复的数据块，达到块级别的Dedup。我们首先就应该想到要处理块层面的内容，那么就一定要在Windows下的驱动链中适当位置插入一个过滤驱动来监测或修改经过的Block IO请求以及向上层通告虚拟的可用容量；然后，还需要一个用来存放针对所有Block所生成的指纹库，这个库可以保存在每个LUN的固定位置，由Filter Driver对其进行隐藏和做数据IO，这样内核文件系统便不会影响到并且损坏指纹仓库了；还需要有一个中央控制模块来实现Dedup的主控。如图18-48所示为笔者自行演绎出来的Dedup模块架构图。

图18-48 Dedup模块抽象示意图

整个Dedup模块的作用原理如下：当初次安装这个Dedup模块之后，Dedup模块便在系统后台运行，Main Control Module根据设定的策略，从一个时间点开始调用IO Thread模块依次读出LUN内的所有Block（Block可以认为设定粒度，比如4KB、64KB、1024KB等）并传递给Hash Handler模块来计算Hash值，算好的Hash值会写入Hash Fingerprints Repository中对应Block的Slot存放备查。主模块会按照一个方向从LUN中读出Block计算，但是如果T1时刻计算完Block n之后，某用户程序或者其他任何上层程序向任何小于等于Block n的最后LBA地址或者发起写IO的话，那么之前所计算的Hash值就会作废无效，因为对应的Block已经被新数据所覆盖了，需要随后重新读出这个Block重新计算Hash。

为了解决这个问题，需要维护一个Delay Block Bitmap来记录当上一次全LUN扫描开始之后到当前位置，这期间系统针对这段位置所发生的写IO所对应的Block的位置，将其在Bitmap中置1。根据这个Bitmap，Dedup模块就会知道指纹仓库中哪些Block的指纹是无效的了。初次全LUN扫描完之后，指纹仓库中会积累相当数量的有效指纹，利用这些有效指纹，Dedup模块开始做真正的Dedup过程。

真正的Dedup过程包含以下4个动作。

(1) Dedup主模块需要处理上层源源不断发生的写IO，Filter Driver实时监测写IO，对于符合条件的写IO，在将其透传到下层的同时，会复制一份将其传递给Dedup主模块，Dedup主模块会实时地针对每个接收到的写IO计算Hash值并且存储到指纹仓库中对应的Slot中，并且在Delay Block Bitmap中将对应的位置0。这样就不会漏下每一个写IO，至此，Delay Block Bitmap中被置1的位不会再继续增加了。如果因为各种原因Dedup模块工作异常或者由于各种限制条件比如CPU负载过高等，Dedup引擎暂停工作，那么每个写IO并不会实时地被计算成Hash值，而引擎转而向Dealy Block Bitmap中记录这些写IO的位置，在系统不忙或者引擎开始工作时根据Bitmap来读出这些Block并计算Hash。

(2) Dedup主模块还需要在后台查找指纹仓库中重合的指纹，一旦找到，便会对所有重复的Block生成一个指针并且保存在On Disk Pointer List中。这个动作交给Block Mapping Metadata Handler处理。这个过程是真正的数据消重过程，也是比较耗费计算资源的过程，所以考虑到系统当前的CPU负载，根据所设定的处理力度进行处理。这个步骤可能有人有疑问，为何引擎不去删除磁盘上的冗余Block呢？这个疑问保留在下文中解释。

(3) 针对Delay Block Bitmap中仍然被置1的Block，从LUN中读入这些Block，计算Hash值存入指纹仓库，并将对应的位置回0，当所有位都被置0后，表示当前的指纹仓库中针对所有Block的指纹全部都是最新的了。

(4) 由于每一个写IO都会被写入存储介质中，所以对于每个读IO，Filter Driver会直接透传到下层驱动从读IO的目标地址读入数据并且返回给上层。这个步骤可能有人也会有疑问，如果每个写IO都被写入介质，哪里能体现剩余空间增加了？这个疑问一并留作下文解释。

这4个动作不断地循环执行，当然，需要注意线程时序逻辑竞争的情况，避免其发生。图18-48中椭圆形组件表示其为一个数据结构，保存在LUN固定位置中，但是Dedup模块执行时会将这些数据全部或部分地载入内存以加快处理速度。

对于操作系统所在的LUN，主机端的Dedup模块无能为力，因为主机操作系统启动过程中难免会对一些Block有写入动作，而此时Dedup模块尚未加载，所以此时指纹仓库中对应的Block指纹已经无效，但是Dedup模块却无法感知这种无效，所以每次重启，Dedup模块就需要重新来一遍全LUN扫描重新生成指纹仓库，得不偿失。所以不能够针对操作系统所在的LUN进行Dedup操作，要想实现，必须使用存储端Dedup模块（见下文）。

对于LUN与LUN之间的数据消重，就需要Dedup模块以所有LUN中的指纹仓库作为一个全局的查找对象，一个LUN中的Block如果与另一个LUN中的Block内容相同，则也可以做到消重。

Dedup是一个后台异步处理过程，它所带来的效益并不是实时就可

以感觉到的。当然也可以设计成实时的Dedup，比如写IO进入后必须先交给Dedup模块进行Dedup处理，如果被判断为重复数据，那么便不会将实体数据写入后端磁盘。这样做有一定的风险性，如果写IO是Write Back模式，那么Filter Driver会在接收到IO请求立即返回成功信号给上层，那么此时就要求Dedup模块必须有某种机制来保证这个写IO在系统故障时也不能丢失，而对于主机端的Dedup模块，唯一办法还是要将其先写入磁盘，或者日志链中暂存，所以依然无法避免磁盘IO，与其这样，还不如保持后台异步方式的好。而如果是Write Through模式，那么前台处理将严重增加IO延迟，影响性能。

提示： 指纹仓库中包含有很多的Hash值，Dedup引擎一般使用Bloom Filter算法来快速地查找当前给出的Hash值是否存在于指纹仓库中。Bloom Filter算法本身需要将指纹仓库中的Hash值本身再次做一系列的处理，这其中包括将其再次Hash。详细步骤就不做过多描述了。

5. Dedup的分类

Dedup的设计总体上可以分为多种类别，作用于不同层次、目标和阶段。

1) In-Band/Out-Band Dedup

所谓In-Band，就是将指纹仓库存放在与所要Dedup的目标实体数据所在的相同位置。比如要Dedup某主机上的若干LUN，那么指纹仓库也放在这台主机上的一个或者多个LUN中固定位置，这样，有多少主机需要实现Dedup，就有多少份指纹仓库存在。

所谓Out-Band，就是将整个全局系统内的所有主机上LUN的指纹保存在一个单独地点，集中存放，集中管理。主机上的Dedup引擎通过网络与这个指纹仓库通信从而存入或者提取指纹。由于Dedup多是后台异步处理方式，所以外部网络所引发的延迟并不会从根本上影响一段时间内的Dedup的效率。

传统设计总是采用In-Band的方式，每台主机上都安装一个独立的Dedup引擎。然而，这只是第一步。有没有可能采用一个集中Dedup引擎来统一管理全局系统内的所有主机上的Dedup操作呢？答案是有。如图18-49所示为将Dedup引擎全外置之后的彻底Out-Band模式架构。

图18-49 Dedup引擎全外置，Out-Band

上图中的架构是一个Client/Server架构，基于图18-48中所示的自演绎的Dedup引擎架构改造。原本运行在每台主机上的Dedup引擎的主干组件全部被外置到一台单独的Dedup服务器主机上，而客户端主机上只安装Filter Driver和一个用于与Dedup服务端通信的Agent。服务端与客户端之间通过某种网络来通信，由于Dedup是一个后台异步过程，对实时IO性能基本上影响很小，所以这个网络采用普通千兆以太网即可。Dedup服务端为每个加入Dedup域的客户端主机维护各自的Metadata，包括指纹仓库、Delay Block Bitmap和Pointer List。

对于每个主机客户端的写IO，会被客户端复制一份通过网络传送给Dedup服务端进行Hash计算和Dedup处理以及Pointer List更新。所以，这个网络上的流量会是比较大的，因此最好使用单独的网卡和交换机而不要与前端交换机混用。另外，向服务端发送写IO数据的过程对客户端主机的原本IO性能影响不大，但部署时需要避免这块网卡与客户端原本的IO适配器处于同一个主机IO总线上。

每个主机客户端的读IO操作都需要查找Pointer List用以确定IO的目标地址所落入的Block是否已被Dedup，如果尚未被Dedup，那么Filter Driver就会透明发向下层直接从底层磁盘对应地址读出数据；如果已经被Dedup，那么Filter Driver会根据Pointer List中的源副本所在地址读出对应的数据部分。但是在上图所示的这种彻底Out-Band模式下，每个主机客户端的Pointer List都被存储在Dedup引擎服务器上，如果每次读IO都会引发一个向外Dedup部服务端的查询请求，那么延迟将会是非常大的，严重影响性能。鉴于此，可以将Pointer List缓存在主机客户端本地，由Agent负责维护以供Filter Driver直接查询，避免通过网络。被缓存的Pointer List需要时刻保持最新，Dedup服务端需要实时地将Pointer List的变更推送到客户端的Agent处。

2) 主机端/存储端/备份介质服务器端Dedup

与其大动干戈地在主机端搞什么In/Out-Band架构设计，不如干脆将Dedup引擎全部放入存储设备之内来运行，主机端不需要做一点改动，完全透明访问所有的Dedup过的LUN。可以看到存储端Dedup架构与主机端别无二致，因为存储设备本质上也是主机。

图18-50所示的架构既可以针对Block存储设备，也可以针对NAS存储设备。当然，在NAS上也可以实现文件层面的Single System Image模式的Dedup。

图18-50 存储端Dedup引擎架构示意图

另外，也可以将Dedup引擎放到备份介质服务器上，所有客户端的数据源源不断地被传输到备份介质服务器上，Dedup引擎对这些备份数据进行消重处理。值得一提的是，备份介质服务器上的Dedup引擎可以

针对不同的备份介质选择前台处理或者后台处理，详见下文。

3) Online/Offline Dedup

Online Dedup是指被Dedup的目标数据为当前的应用系统正在访问的数据。由于Dedup过程对目标LUN有一定额外的IO消耗和IO延迟，包括初始化Dedup时读取Block进行Hash计算的过程以及针对每个读IO都需要额外的查询Pointer List的步骤，所以Online Dedup对生产系统的IO性能有一部分影响。

Offline Dedup针对的Dedup目标则为那些已经脱离在线业务生产系统访问的离线数据，比如，备份之后的数据。针对这些数据所做的Dedup操作不会对生产系统有影响。

4) 局部/全局Dedup

所谓局部Dedup是指，Dedup引擎只在进入的某段数据范围内做Dedup。比如单位时间内有一串128KB字符进入，如果是局部Dedup，则引擎会将这128KB数据分割成比如4KB大小的Block，算出每个Block的Hash，然后只比对这些Block之间有没有冗余数据的存在，如果有，则Dedup其中冗余部分，只写入唯一的数据部分。

所谓全局Dedup是指，Dedup引擎会对进入的数据切割成比如4KB的Block，算出每个Block的Hash之后，Dedup引擎会去系统全局指纹仓库中查找匹配的指纹，这样的话，去重比率将更大。全局Dedup的另外一个含义是，可以做到跨设备的Dedup，也就是说多个设备共同形成一个大的存储池，指纹仓库存储着所有设备上的数据指纹，实现跨设备的全局Dedup。

局部Dedup的本质更加接近于压缩，而全局Dedup才是真正的Dedup。

5) Pre-Process/Post-Process Dedup

Pre-Process意即前台处理，Post-Process则表示后台处理；Pre-Process又被称为In-Line Process，Post-Process又被称为Out-Line Process。前台处理的步骤如本小节4部分结尾所述。前台处理可以保证系统时刻处于完全Dedup状态，任何时刻系统中都没有冗余数据存在，但是相应地它也增加了处理负担和延迟，对Write Through模式的写性能有严重影响。

而Post-Process，即后台异步处理，根据设计的不同也有不同的影响。比如某种设计选择完全的异步处理，即写IO在进入系统时，Dedup模块只在Delay Block Bitmap中记录其位置，而不计算Hash值，任凭它们被写入存储介质。当Dedup引擎因为某个条件被触发时，比如时间、剩余空间等，引擎将根据Bitmap来读出对应的Block并且计算Hash，然后进行消重。但是这种做法需要耗费额外的读IO消耗。在本小节4部分中的首选设计方式并非完全异步处理，其步骤中有一定的同步处理过程，即每进入一个写IO，Dedup实时地计算其Hash值并且保存到指纹仓库，只有在发生Dedup引擎异常或者系统CPU覆盖过高时才转为异步处理。这样做的话，指纹仓库中大多数时刻都是最新的指纹，无须回读过程，任何时候Dedup消重子功能被触发，引擎只需要在指纹仓库中查找相同的指纹即可。

综上所述，在Online的数据，总是应当使用Post-Process Dedup。对于Offline数据，由于不会影响生产系统性能，所以可以选择使用Pre-Process。一个典型的处理Offline数据的角色就是备份系统中的介质服务

器，当连接到介质服务器的存储介质为物理磁带等不可快速随机寻址的存储介质的时候，采用后台Dedup处理是不恰当的，此时只能在数据被写入磁带之前进行Dedup处理。这方面详情见下文分析。

提示： 以上的类别划分于不同层次上，每个类别之间可以互相组合，比如“主机端的Out-Band的Offline的Post-Process全局的Dedup”。

6. Dedup的去重比率影响因素

1) Block大小的影响

毫无疑问，Block粒度越小，发生数据重复的几率就越大，去重比率就越高。如图18-51所示，同样一段数据，如果按照Block=8KB来计算一个Hash值的话，那么图中上面的两个Block的Hash值不同，则这两个块任何一个也无法被去重。而如果Block=4KB，那么其中的就有一个Block与另外一个Block发生重复，其中一个就可以被删掉。然而，取得高比率就要付出维护复杂度增高、性能降低的代价，孰轻孰重，可各自把握。

图18-51 不同Block 大小对去重比率的影响

2) Block边界的影响

如果数据并不是按照图18-51中所示的那样排列，而是按照图18-52所示的排列方式，而且又不加以特殊处理的话，那么这个例子的去重比率就是0。

图18-52 边界位移导致的去重比率降低

上图左边所示的例子中，只因为Block边界错开了一个字符，却导

致这段数据的局部去重比率为0。右边所示的为两串字符串数据，只因为首部插入一个字符，导致不管Block Size为多大（除非一个字符的粒度），这个全局的Dedup比率也为0。

这种问题是现实中很有可能发生的，比如用户在某个文档中插入一个字符，然后另存成一份新文档，那么这个新文档对应的底层Block内容全部前移一个字符，新文档和旧文档虽然只差一个字符，但是传统的Dedup却无能为力。

为了解决这个问题，一些厂商实现了位移技术，即针对待处理的数据，先从二进制流上进行简单抽样比较以确定是否发生了这种位移现象，通过比对结果确定发生位移的偏移量，然后使用特殊的处理方法，截掉首部插入导致整体位移的数据段，单独存放并且记录特殊的元数据信息，剩余的数据部分再按照常规的分块计算Hash然后Dedup处理。在提取数据时，按照当初存放的元数据信息以及截取的数据段以及唯一Block副本共同组合还原成原来的数据。

实现这种可变长度时的算法都是有固定模式的，各个厂商可能使用的算法大同小异。基本上都是按照某窗口对目标数据进行扫描，比如首先错位1字节，如果没有发现冗余块，则错位2字节扫描，依此类推，直到错位到一个最大窗口，比如16字节，之后如果还没发现一定比例的冗余块，则放弃这次扫描，认为其无冗余。

这种方法目前被厂商称为“可变长度”的Dedup技术。传统的方法被称为“固定长度”的Dedup技术。可变长度Dedup技术只对局部Dedup有效，因为全局Dedup的情况下，要进行边界位移，就必须提前感知到当前进入的数据与已经存储的数据中的哪些是重复的以判断在哪里位移，而这是不可能的。

7. 深刻理解Dedup的作用层次，真的Dedup了么

现在是解释本小节4部分中的疑问的时候了。我们要时刻牢记，Dedup是为了什么而生的呢？当然是为了节省底层存储空间，可以把节省出来的空间再利用而生的。那么也就是说，如果不能节省出底层空间，或者节省出来的空间并不能够再利用，那么所做的任何事情、任何技术、任何设计，都将是没有任何意义的。

回顾本小节4部分中的几个Dedup步骤。当Dedup引擎在指纹仓库中找到相同指纹之后，它不去磁盘上将冗余的Block抹掉，而只是将这个信息更新到Pointer List中。有人会有疑问，如果某个Block被Dedup了，难道不需要去数据卷中对应的位置将其标记为指针么？而冗余Block依然处于卷中未被删除，那么怎么会节省出空间来呢？其实抱有这个疑问的人尚未理解一个基本原则：在主机上，是谁来最终决定LUN的剩余空间？这个问题在本书中曾经不止一次提到过，LUN一层根本无法感知文件系统的剩余空间，所以，即便是LUN中的冗余的块被Dedup了，但是这种Dedup之后所节省的空间，也只是由Dedup引擎自己知道（通过统计Pointer List中的条目）。但是上层的文件系统已经认为这个块被某文件占用了，所以不会把它当做剩余空间来对待，也就是说，你只能眼睁睁地看着实际的节省出来的空间不能够被文件系统使用。这么说，Dedup根本没意义了？Dedup不是应该节省出一部分数据空间的么？先别急。

结论1： 任何不能被文件系统感知到的回收的空间，对于这个文件系统来讲是没有意义的。同样是回收空间，但是De-Provision的做法与块级Dedup有着本质不同。De-Provision是运行在存储端的，它使用某种方法将主机端文件系统中的剩余空

间映射到LUN，在LUN所占据的对应的这块空间上“打孔”，也就是“Hole Punching”，这些孔洞就变成了剩余空间，那么同一个存储设备上的其他已经存在的Thin LUN需要占据空间时，就可以利用这些空间，而也可以在存储端利用这些空间创建新的Thin LUN，甚至常规LUN。De-Provision与Thin-Provision紧密结合起来达到共生。

不知道大家现在是否看出一点端倪来，仔细思考一下个中关系。怎样才能使主机端的块级Dedup变得有意义呢？答案是，要么老老实实在地让文件系统感知到剩余空间的变化，要么就让主机端块级Dedup实现与存储端块级Dedup类似的功能。

方法1： 让主机端文件系统感知到空间的节省。你会发现，对于块级Dedup来讲，这种方法永远行不通。为何呢？因为块级Dedup根本没有考虑一点点FS层的逻辑，它与De-Provision的本质是不同的。所以，文件系统始终认为底层被Dedup的块是有文件在占据的。但是对于文件级别的Dedup，比如SIS，是完全可以做到的，SIS会作用在文件级，会将文件中冗余的数据切掉，而文件的元数据中的文件长度保持不变，但是底层占用空间却变了，这些修改都可以被文件系统感知到，所以剩余空间也就增大了。当文件系统发起对文件中被Dedup的内容访问的时候，SIS的底层过滤驱动会截获对应的访问IO请求并且读出唯一副本并返回给上层。

方法2： 模仿主机端对LUN的管理和Thin/De-Provision。毫无疑问，要做到此，就需要在主机端的LUN之上再建立一层空间管理层，也就是卷管理层，在卷管理层之内实现对LUN之上的卷的Dedup而不是对LUN来Dedup。这样做了之后，卷管理层

便会感知到哪些卷实际占用了多少物理空间，而节省出来的LUN中的空间便可以建立其他的新卷。当然，如果在这个卷管理层内同时实现Dedup、Thin/De-Provision的话，那么其节省的空间将会更多，存储利用率将会更高！

提示： ZFS既是一个LUN/卷管理层又是一个文件系统，它相当于把外部智能存储设备具有的功能集成到了主机之内，类似于Symantec Storage Foundation。ZFS对LUN的管理是非常灵活的。并且ZFS已经实现了卷级别的Dedup。

结论2： 一切在主机端运行的、块级的、针对LUN的Dedup，根本无法达到节省和利用底层存储空间的目的。

那么主机端针对LUN的块级Dedup是否完全没有意义了呢？非也。比如，在备份这个LUN的时候，主机端就可以不备份被Dedup的Block，节省备份数据的存储空间。如果备份是通过网络将数据传送给备份介质服务器的，那么还可以大大降低传送的数据数量，降低备份所需的时间。

结论3： 主机端块级的、针对LUN的Dedup，只有在备份整个LUN时才会表现出节省传输数据总量、降低备份窗口的作用和意义。

综上所述，块级的Dedup最好是直接作用在最终的存储端，因为我们最终要节省的就是物理磁盘的空间，而物理磁盘直接归存储设备管理，所以没有理由不在存储端来实现Dedup。然而，数据存储技术不但包含如何存储数据，还包含数据在系统中的传输，比如网络数据备份系统。

8. 网络数据备份系统中的Dedup

不管是否是Frontend-Free模式的备份，既然要备份，那么一定牵扯到数据从源流向目的，而目的就是备份介质服务器，备份介质服务器再将接收到的待备份数据写入底层的备份介质，比如磁盘、磁带等。所以，不管是哪一层的Free，数据传输过程总是无法避免的。而如果需要备份的数据中包含了很多冗余的内容，那么在传输的时候就会产生浪费，增加了额外不必要的数据传输时间。

而如果在数据传输之前，将待备份的数据进行Dedup操作，在传输过程中只传输唯一的数据副本以及对应的Pointer List等Metadata，那么就会大大降低需要传送的数据数量，就像上一节中的结论3所指出的那样。当然，结论3中指出的在主机端实现LUN的Dedup在备份时会发生效果，但是这必须要求Dedup引擎与备份恢复系统配合，比如备份服务器发起备份操作时，Dedup引擎必须介入，否则的话将会以传统方式备份带有冗余的数据的整个LUN。所以，在实际产品中，Dedup模块一般都是作为备份软件的一个子模块来存在的，安装在客户主机之上。

然而，很少有直接备份整个LUN的，大多都是备份文件，所以此时主机上的SIS模块便可以与备份系统相配合。

也不一定非要在主机端实现Dedup，有时候宁愿牺牲一些数据传输时间，也不希望客户主机过多的介入从而影响主机性能。此时，就可以在备份数据离开客户端主机的下一站，也就是备份介质服务器处进行Dedup操作，这样的话，数据到了备份介质服务器之后都将被存储为一个打包的映像文件，所以也就不必考虑文件级还是块级了，统一使用块级Dedup以提高去重比率。在介质服务器上实现Dedup可以有两种选择：前台Dedup和后台Dedup。

如果备份介质为磁带，则必须以前台的方式来实现Dedup，因为磁

带是非高速随机寻址介质，不适合后台的Dedup。在前台Dedup过程中，可以采用局部和全局两种模式的Dedup。如果采用局部模式，则介质服务器上的Dedup引擎对接收进来的数据实时切割成Block然后计算Hash值并存储到指纹仓库中存放，第一个Block无条件直接写入介质，第二个Block进入后先计算其Hash值并且与已经存在的指纹进行比较，如果匹配，那恰好不用写入介质，只需要将这条匹配信息记录在Pointer List中即可，后续的Block的处理以此类推。这样，就可以针对这个备份流自身进行Dedup，也就是局部Dedup。如果采用全局的Dedup方式，则介质服务器上的Dedup引擎对接收进来的数据计算Hash值后，会将这个Hash值与之前曾经被备份过的数据对应的指纹仓库中的Hash值进行比较从而Dedup，也就是说进入的数据在全局范围内是没有冗余的，但是全局模式对Dedup引擎的效率以及介质服务器的处理能力要求比局部模式要高。

介质服务器都有本地的硬盘，所以也可以先将备份数据流暂存在本地硬盘的某处作为缓冲，然后Dedup引擎后台处理，之后写入磁带，也就是使用D2D2T方式。前台操作可能会让系统应接不暇，降低整体性能。另外，操作磁带设备与操作磁盘设备相比，会耗费更多的CPU周期，所以前台Dedup处理，同时后台还要写入磁带，系统整体性能将会非常低下。因此D2D2T的方式是Dedup应当采用的方式。

如果备份介质为磁盘或者基于磁盘虚拟出来的磁带（VTL），则可以采用后台Dedup方式，因为此时可以快速定位和读出需要的Block。在这种模式下，介质服务器首先将收到的数据流按照常规方式直接写入磁盘或者VTL。当Dedup引擎触发时，再从对应的目标将数据读出计算Hash并且做Dedup处理，然后再写到新的介质空间，原有的备份数据集删除以腾出空间。

在进行数据恢复时，介质服务器上的Dedup引擎必须介入，从所记录的Pointer List以及数据唯一副本，共同生成一份完整的数据，然后恢复到客户端主机上。

当然，也不一定非要在介质服务器上实现Dedup。在外部备份存储介质设备中比如磁盘阵列、磁带库、VTL上一样也可以实现Dedup，这里便是数据的最终归属地了。在这两处实现Dedup，其本质原理都是一样的，所以就不做过多介绍了。

综上所述，Dedup与备份是绝对不可分割的，Dedup在数据备份恢复的时候能够起到重要的作用。另外，作用于Online存储系统上的Dedup可以节省存储空间的浪费，腾出空间以做它用，也是Dedup发挥重要作用的一处。

9. Dedup思想的其他应用

增量备份、差量备份，通常是在应用程序层来控制的，应用程序来负责备份它所操作的文件以及记录每次备份的时间点以及文件所变化的部分。备份软件在这里起到的作用只是配合应用程序将其生成的数据流导向存储介质中保存，同时管理和维护存储介质，以及制定备份任务计划等功能。当然，备份软件也可以直接将整个LUN备份下来。

然而，备份软件自身如何能够实现LUN级别的增量或者差量备份呢？备份软件虽然不能够感知各种应用程序所生成的文件中变化的内容部分，但是它可以记录底层的Block变化，将上一次备份整个LUN之后，这个LUN中所发生的所有写IO对应的Block，按照位置记录在一个Bitmap中。下次触发备份时可以只将Bitmap中被置1的Block备份下来即可，这样就实现了差量备份。

如果要想实现增量备份，则可以将上一次的Bitmap封存，新建一份空Bitmap来保存自上一次增量或增量备份之后LUN中变化的Block，当再次触发增量备份时，将这份新Bitmap中被置1的位对应的Block备份下来，并封存此Bitmap，创建新Bitmap，依此类推。在进行恢复时，会根据需要恢复的备份点，将这个备份点之前的所有Bitmap做一个OR操作，合成一份完整反映需要恢复的Block的Bitmap，然后首先将初次全备份的数据覆盖到目标上，再根据这份完整Bitmap和每次增量所备份的Block集，提取对应的Block然后覆盖到目标对应的Block位置上。这种方法可以大大减少每次需要备份的数据量，但是却不能减少每次恢复时所恢复的数据量。

上面的做法可以从一定程度上消除一些不必要的数据块的恢复，但是并不能彻底杜绝传输一些不必要的冗余数据块，因为备份软件是无法感知到多次增量备份的数据块是否有冗余数据存在。如果待恢复的目标处已经存在对应的数据，比如某个LUN，由于用户误对LUN进行了格式化操作导致数据丢失，需要恢复，那么这个LUN会被从备份介质中完全读出并且覆盖，纵使当前损坏的LUN中数据几乎与待恢复的数据相同（格式化只是抹掉初始文件系统入口之后新创建一个而已），所以，这就会产生资源浪费。如果数据是通过网络进行传输的，那么传输多余的数据块就会增加不少恢复时间，增加RTO。

如果用Dedup的思想，在进行恢复之前，先对待被覆盖的数据和待恢复的数据各自计算Hash，通过比对找出相同的Hash，然后只覆盖有差别的Block，这样的话，在一些情况下（比如上面的格式化例子），将会大大降低RTO。某些产品已经可以实现这个功能，称为基于Hash的增量恢复。

Dedup技术也可以帮助在数据容灾过程中降低需要传送的数据的整

体数量，比如对某个待传输到灾备站点的数据流，可以对其做局部Dedup处理，大大减少冗余数据的发送。

10. Dedup功能在产品中的嵌入和表现形式

上文中曾经描述过，数据存储技术包含如何存储数据以及如何传输数据。那么Dedup就可以作用在数据的存储点，也可以作用在数据传输的通路上。前者的模式在上文中已经有很多描述。对于后者的模式，则可以在数据传输通路中设置一个网关，在这个网关设备上部署Dedup模块，任何通过这个网关的数据流都被Dedup了，前端的汹涌大河，被Dedup了之后就变成了涓涓细流。细流的尽头是备份介质服务器，介质服务器将细流写入慢速的磁带来保存。这个关口大大降低了后端介质服务器需要写入的数据量，降低了备份窗口。

目前的Dedup产品，有些运行在主机端以降低备份时所需要传输的数据量，有些运行在存储端以达到节省空间的目的，有些运行在数据传输的通路中来消除冗余数据，有些运行在介质服务器上来降低写入后端介质的数据量，有些则运行在数据的最终归宿——磁带或者VTL上，对数据做最后的Dedup。

11. Dedup存在的问题和风险

Hash冲突当然是第一大风险。第二大风险就是数据没有冗余，一旦源副本受损，则一损俱损。但是对于第二个问题，如果Dedup用于Online数据中，那么确实会有这种风险。但是如果针对于Offline的备份数据，那么似乎并不是一个真正的问题，因为备份已经是Online数据的冗余，虽然冗余一份相比冗余多份来讲保险系数要低，但是源副本受损，很大程度上是因为整个介质受损，这样看来，一损俱损也是逃不过

的。

Dedup对数据IO性能方面有一定影响。在一个备份系统中，在没有引入Dedup之前，所备份的数据都是实实在在的物理数据，每次备份下来的数据基本上都会在物理位置上被连续存放。而一旦引入了Dedup，那么多次备份的数据中的冗余数据就会被用指针替代，那么就会产生“孔洞”，使得本来物理上连续的数据变得不再连续。这样，当要从被Dedup之后的数据中将某个备份集读出来的时候，此时在底层就属于一种随机IO了，有时候从Dedup的磁盘介质中恢复数据的过程，甚至比从磁带中还要慢。这种效应会随着时间的流逝而越来越严重，就像文件系统中的文件碎片一样。

Dedup在写入数据的时候基本上有两种方式：reverse referencing和forward referencing。前者的做法是，当在系统发现一个冗余数据块时，系统会保留之前被存储的旧数据段，同时在旧数据段处写入一个新指针，而不是将新数据写入磁盘；后者的做法是，系统会将新数据段写入磁盘的连续的物理位置上，然后将旧数据删除同时将旧数据替换成一个指向新数据段的指针。对于前者，也就是不断为新的冗余数据段写入指针的做法，可能会导致最新的数据对应的碎片不断增加；对于后者，也就是系统不断地写入新数据而删除旧数据同时替代为指针的做法，则可能会使最新备份的数据物理上保持连续，但是旧备份对应的数据在物理上却变得越来越随机。

因为涉及到更大的磁盘写入数据流，forward referencing模式如果在前处理模式的重复数据删除系统中使用的话，会严重影响性能，所以它只在后处理模式的重复数据删除系统中使用，这种模式的数据存储方式可以让最新的备份数据恢复起来更快，而且这也是最常见的情况。但是Forward referencing模式却会引来额外的IO开销，比如它不但需要将冗

余的数据先写入磁盘而不是写指针，而且随后还会查找旧的冗余数据进行删除操作。

除了使用Forward referencing模式来确保最新被备份的数据具有较高的读取速度之外，有些厂商还在Dedup系统内实现了类似于碎片整理的操作，或者有些厂商干脆对最新一次的备份不进行任何Dedup操作，以确保它的恢复速度。

不管是Thin、Tier，还是Dedup，其实它们都是很早就已在其他领域中实现的技术，如今这些技术被用在了网络存储领域中，来回翻炒以便挖掘价值和发展。

12. Dedup之后再压缩

为了获得最大的空间节省比率，还可以在对数据进行Dedup处理之后，再次进行压缩操作。有人可能会不太明白，既然Dedup的节省效果比压缩高得多，那么为何Dedup之后还要压缩呢？因为正如前文所述，压缩有一定的窗口，只可以在小范围数据长度之内进行数据消重，而Dedup则可以在更大的范围甚至全局范围内实现消重，但是它也有一定的最小粒度，比如4KB、16KB等，它可以将多份相同内容的颗粒进行消重只保留一份，但是却不可能对这一份颗粒内部的冗余数据再进行消重，那么此时传统的数据压缩算法就可以派上用场了。如果目标数据是文本等冗余度较大的类型，那么Dedup之后再压缩，依然可以获得将近20%~30%的空间节省。这是相当可观的，但是也有一个最大劣势，就是在压缩数据时以及读取时的解压缩过程中，会非常耗费CPU资源。所以这种手段一般常用在VTL等备份环境中，由于传统的磁带机和机械磁带库中的驱动器均具有压缩功能，所以这种压缩不需要额外引入第三方模块，但是对于VTL来讲，就必须引入一个压缩软件模块了。也就是

说，一台VTL上可能同时具有Dedup以及压缩模块，有的产品还引入了硬压缩卡来分担主CPU的负担。

18.3.5 磁盘数据一致性保护及错误恢复

1. 坏扇区重定向

数据在系统中流动的时候，随时可能会发生畸变，比如某个位原本应该是0，却被畸变为1。这种情况尚没有办法解决，但是有些存储系统为了保证至少数据从硬盘被读入内存这个阶段中不会畸变，引入了一些额外的校验值来对读出的数据做校验，如果一致则不动作，一旦发现不一致，则通过某种手段来恢复之前的数据。比如通过校验值来纠正，或者通过RAID层面的Parity来恢复整个Segment上的数据。

其次，当遇到物理磁盘坏道的时候，也需要一些处理措施。磁盘自身会有坏扇区/坏道重定向功能。比如某时刻磁盘接收到针对某扇区的写操作，但是却发现此扇区已损坏，无法写入，那么此时磁盘会在其盘片的保留区域分配一个扇区作为这个坏扇区的顶替，将内容写入这个新扇区，并且在映射表中增加一个条目以便后续所有针对这个地址的操作都重定向到替代扇区。但是对于读操作，如果接收到主机针对某扇区的读操作而发现此扇区已损坏无法读出，那么此时磁盘只能去报错，因为此时数据已经读不出来了。

在磁盘阵列中，如果某个RAID组中有磁盘报扇区损坏，那么控制器此时可以从RAID组中其他磁盘同一条带中的数据做XOR计算来恢复出对应Segment的数据。一方面控制器可以将读出的数据返回给外部发起IO的客户端主机，同时，控制器将这块数据再覆盖写入有坏扇区磁盘的对应Segment，当这块磁盘试图写入这个坏扇区时发现无法写入，此

时就不一样了，磁盘会自动将这个坏扇区重定向到保留空间的某个好扇区中，这个动作对上层是透明的。写入之后，需要再次读取这个地址的内容来验证是否扇区重定向动作真的成功了。利用这种迂回的方法，阵列控制器可以将某块磁盘的寿命延长。这种做法在Linux下的软RAID中就有实现，称为Bad Sector Remapping（BSR）。

此外，磁盘自身的保留区域容量也是有限的，如果由于坏扇区或者坏道过多而导致连这些保留区域也耗尽的话，那么这块磁盘就不能再自动重定向了。那么此时就可以宣告磁盘完全故障了么？并非如此，还可以继续延长它的寿命。如果遇到这种情况，那么此时RAID管理层可以发挥作用了，可以这样设计：让它先向LUN管理层查询一下当前这个坏扇区地址是否是被某个LUN占用的，如果确实是被某LUN占用，那么RAID层立即先用其他磁盘上同一条带中的数据将这块丢失的数据算出来，然后将整个条带搬移到RAID组中剩余的、未被任何LUN所占用的空间内，并且做好指针映射记录，或者只重定向这个损坏的Segment（一个条带占用一块盘上的空间称为Segment）。这样，今后的所有IO均不会再落入到这个损坏的扇区或者此道中，从而将磁盘的寿命压榨到极限！

2. DIF（Data Integrity Field）数据一致性保护

数据从被应用程序生成，一直到被存入硬盘，期间需要经过多次内存复制、流经系统总线、进入IO总线、流出IO总线到IO芯片，然后流出IO芯片到外部存储网络，比如SAS、FC、SATA等，然后进入磁盘缓存、后流经磁头臂，被转化为磁信号，然后被磁化到盘片中。由1和0两种状态组成的数据，在所有这套路径上的每个部件中流入、流出的时候，难免会发生一些畸变，比如由0变为1或者相反。发生畸变的原因有

多种，外界扰动、软件bug等等，都可能造成数据畸变。不仅在外部传输的时候会发生畸变，就连磁盘要写入盘片的时候，也有一定几率发生。

上述数据畸变，都不会被上层感知，所以这类畸变统称Silent Corruption。虽然上述数据畸变发生的几率，按照常理来讲应该非常低，但是实际却并不是很乐观。NetApp曾经对150万块在线SATA硬盘做了长达32个月的跟踪，结果令人震惊，0.66%的SATA盘都出现了数据静默损毁，FC盘的数据静默损毁比率则为0.06%。虽然有些阵列后台都有磁盘扫描功能，可以检测到数据域Parity的不一致，但是这种扫描属于后台异步方式，并不能100%检测到不一致，据统计仍有13%的不一致并不能被检测到。

静默损毁一旦发生，则后果是比较严重的，基本都会造成数据丢失。本书之前章节介绍SSD时，曾经说过SSD中的Cell失效率很高，需要引入ECC纠错码机制来纠错，比如每512B的内容引入32b的纠错码，就可以纠正这512B+32b的数据中任意32b的畸变。本书第3章中可以看到一个机械磁盘扇区的全貌，扇区尾部有一段ECC校验码用来纠错。ECC可以侦测和纠错大部分的数据畸变。但是，却不是万能的。

比如典型的三种问题就是Lost Write、Torn Write以及Misredirect Write。Lost Write是指对于某个扇区，磁盘根本就没有去写入对应的扇区就向上层报告写入成功了，此时这个扇区中的所有数据依然是未覆盖之前的，也是一致的，ECC无法发现，但是数据确实丢了。Torn Write则是指磁头只部分写入了对应的扇区，就返回成功信号了，此时视已经覆盖的数据量的多少，如果太多，那么有限的ECC位不能够纠错如此多的不一致数据，就算覆盖的比较少，ECC可以纠错，那么纠错之后，也会是这个扇区未覆盖前所对应的数据，数据依然丢失了。Misredirect

Write是指，磁头在写入某个扇区内容的时候，定位不准，将内容写到了目标扇区旁边的扇区中，或者由于Firmware等的bug，写到了压根与目标扇区没有任何物理近邻关系的位置相隔很远的扇区中，接着返回成功信号，此时ECC无能为力。Lost Write发生的原因尚无准确的结论，Misredirect Write的原因大部分是外界扰动，Torn Write的细节原因，我也无法知晓。这些限于微观状态的动作，其原因定位非常困难。

为此，需要一种更加智能的一致性侦测手段。这种手段就是在数据区增加一个DIF（Data Integrity Field，一种由T10制定的标准）区段用来记录一些上层的信息而不仅是校验值。对于FC或者SAS磁盘，厂商一般在低级格式化的时候将每个扇区格式化为520B而不是常规的512B（SATA盘使用512B扇区）。多出来的那8B就是DIF。SATA盘为何不低格为520B扇区呢？这么做是有原因的，因为SATA盘一般不用于企业级存储系统中，而FC与SAS则是面向高端应用，需要严格保证数据一致性。但是近年来随着数据海量增长以及成本要求不断降低，SATA盘越来越多地被用于企业级存储系统中，但是其512B扇区的设计依然未变，并且SATA盘稳定性相对FC与SAS更低，所以也迫切需要采取手段来保护SATA盘数据的一致性。我们下面将会讨论这一点。

如图18-53所示为DIF区段在整个扇区中所处的位置以及其内部细节结构。其中Guard字段为2B的CRC校验字段；Application Tag这个2B的字段则是可自定义的任何信息，比如这个块属于哪个LUN，或者属于哪个文件系统、哪个目录甚至哪个文件ID等，都可以，由阵列设计者而定；4B的Reference Tag字段则为本扇区对应的LBA地址。

图18-53 DIF区段的结构

DIF区段会随着每次扇区的IO请求一同被读出或者写入。

下面我们就来看一下DIF是如何帮助系统侦测到LW、TW以及MW的。

（1）DIF如何解决Torn Write问题

先说一下TW，也就是某扇区中的数据只被写入了一部分就返回了，这种情况下，扇区中当前DIF字段中的CRC校验码是与数据区数据计算出的校验码不一致的，当这个扇区被读出时，底层将执行一致性校验，此时便可以侦测出数据不一致。那么后续该怎么解决呢？负责一致性的模块此时需要将这个错误上报到上层，比如RAID层，此时RAID层先读出这个扇区所在条带中的所有Segment数据，依次做检查，如果其他所有扇区数据都一致，那么此时RAID才可以，才敢使用这个扇区所在的RAID条带上其他成员盘以及Parity盘上的Segment一起来做XOR反运算，从而得出这个不一致的扇区原本的内容，然后再次覆盖一遍，从而恢复了这个扇区原本应有的数据。而如果RAID在检查的时候发现这个条带中出现了多个位于不同Segment中的扇区的内容自身CRC校验不一致，那么RAID此时就傻眼了，因为此时相当于多盘失效，已经无法重算出正确数据了，只能向上层报错。

（2）DIF如何解决Misredirect Write问题

再看一下MW。假设某时刻系统向LBA1处进行了写入操作，但是这份数据却被写到了LBA1000中。那么现在的状况是：LBA1000处之前的数据丢失了，需要找回，LBA1处当前的数据并不能使用，因为是一份过期的数据。此时，系统面临着很复杂的境况，即LBA1与LBA1000所在的条带都是不一致的（Raid层下发IO时是一致的，但是磁盘自己写偏了，实际上当然就不一致了），RAID校验层可以通过计算来察觉到这种不一致，因为对数据区的XOR结果根本不会等于Parity区，但是

RAID不会去主动实时的校验每一个条带的，甚至当设备在读出整个条带数据到内存里之后也不会主动做条带校验，因为这样太耗费资源。有些设备在低负载时可以触发主动的后台一致性Parity扫描检查来尽量发现这些不一致，但终究只是做到了“尽量”。下面例子中会举出系统如何“埋雷”的例子。

如果本次MW发生之后，紧接着的下一个针对LBA1或者LBA1000所在条带的其他LBA（非LBA1和LBA1000）任何一个的IO是写IO的话，那么就可能会出现连环污染或者数据永久丢失，而且上层根本无法感知到静默丢失。比如，本次MW发生之后，又有针对LBA1处的正常写入操作，那么如果RAID组为校验型RAID（比如RAID 5），那么此时系统会利用旧LBA1的数据、新LBA1的数据以及旧Parity共同算出新Parity（如果恰好为针对该条带的整条写或者重构写则不会出问题，本例假设为读改写），此时这份新Parity依然是错误的，相当于把错误发生了转移，本来是LBA1是错误的，现在转移到Parity上去了，当然，上层是无法区分到底是LBA1还是Parity有问题，此时如果进行整条带校验计算，会发现仍然是不一致的。但如果是重构写，则会修正该条带之前的错误，也就是第二次写入的LBA1会覆盖之前的原始旧LBA1（第一次写入的LBA1此时依然处于LBA1000处），同时读出该条带剩余的Segment，共同计算Parity后覆盖旧Parity，此时整条带是一致的，LBA1的错误被新LBA1的数据修正了。所以这便成了一个不可预知的状态，完全看运气。但是如果在系统未发现条带不一致之前，针对LBA1或者LBA1000所在条带的其他LBA（非LBA1和LBA1000）的任何一个IO是写IO，而且是读改写的话，那么这份错误的LBA1或者LBA1000的数据便被“合法化”了，因为此时该条带是可以被校验通过的。这颗“地雷”就永远被埋藏于此了，上层应用此时只能使用这份逻辑上错乱的数据，而绝对不会知道正确的数据是什么。这种情况就被称为Parity pollution，

即校验值污染。

通过上述过程的分析我们可以得出结论，也就是如果有一种方法可以让系统感知到发生了MW，那么在特定条件下，系统可以成功纠错。每个扇区的DIF区段里的reference tag记录了该扇区所属的LBA号，系统在内存里会为每笔IO数据贴上其“应该”对应的LBA号，这样就可以让系统感知到MW现象了。比如，当MW发生之后的第一个针对LBA1000的请求为读请求或读改写请求（读改写会先读出LBA1000原有数据），总之系统读出了LBA1000里的数据，此时系统发现该扇区DIF区段里的LBA号竟然是LBA1（之前被磁盘自身写偏）而不是LBA1000，那么就知道了发生了MW，系统此时会立即判断是否“还来得及”，还来得及做什么？当然是纠错。之前提到过，污染是可以被传递的，已经被传递的污染是不可挽回的。比如，假设在系统发现LBA1000里的数据其实应该属于LBA1之前，该条带的其他Segment没有发生写入操作，那么此时系统发现了MW之后，便可以利用该条带其他Segment来校验出LBA1000之前被错误覆盖的数据，判断是否该条带其他Segment曾经在MW之后发生了写入的办法就是计算该条带当前的校验是否通过，如果不通过，证明尚未有写入操作，也就是污染未被传递；如果校验通过，则可以判断发生了污染传递，此时系统无法判断从MW发生到被发现之间这段时间之内系统对该条带上哪个Segment做了写入操作（如果能够判断出是对非LBA1000做了写入，那么还是可以挽回的，但是系统判断不出来），LBA1000之前被错误覆盖的数据就无可挽回了。

但是，上述的判断方法成立的前提是发现某扇区的确出现了MW，也就是对应的reference tag中的LBA号与当前其所处的LBA号不同。如果未出现MW，仅仅是校验无法通过的话，上述判断就不能适用。比如这个场景，MW发生之后，接着收到一个针对LBA1000的正常的写操作，

此时原本应当落在LBA1的数据就会被永久覆盖掉并且永远找不回来。同时，如果是读改写，则会将逻辑错误传递到Parity上去，并且每个扇区的reference tag中的LBA号与其所处的LBA也是对应的，结果就是系统即便发现了该条带不一致，也永远不会知道其实该条带的数据Segment此时已经都是一致的了，系统此时会判断出并没有发生MW（其实是发生过但是没来得及被发现之前就被正确的LBA1000扇区内容又覆盖写了），只能向上层报错。

假设我们运气很好，系统挽回了LBA1000的旧数据并做了恢复，那么这个原本应该覆盖到LBA1的扇区，此时是否可以顺水推舟的覆盖到当前的LBA1处？一样，也需要对LBA1所在的条带判断是否发生了污染传递，如果未发生，则可以覆盖，如果已经发生，则不能覆盖，因为系统无法判断出是否LBA1在MW发生和被发现这段时间内是否发生了写操作，不能擅自覆盖。

如果无法纠错，则系统直接报unrecoverable error，告诉上层说这个错误我恢复不了了，你该怎么办就怎么办吧。此时上层可以使用比如恢复数据、快照回滚等方法来恢复数据，大不了几天的活白干了，重新处理计算一遍，也不能明知系统内有地雷而依然向上层掩盖这个事实。

静默损毁属于定时炸弹，一旦被损毁的数据属于高精尖行业比如卫星导弹，后果不可设想，比如卫星偏离轨道之类。

（3）DIF如何解决Lost Write问题

Lost Write（LW）算是最难解决的问题之一。因为LW在现场根本不会留下任何犯罪证据。TW会留下扇区自身CRC校验不一致的线索，MW会在错误重定向之后的扇区留下表里不一的线索，而LW呢？大家

想一下，LW是指磁盘根本没有去碰对应的目标扇区，就向上层返回写入成功了，此时，除非事先就知道发生了LW，否则你根本无法知道是否发生了LW。然而，一点办法都没有了么？非也。系统底层肯定是无法感知到LW，那么就只能靠上层来解决。DIF区段中还有一项叫做Application Tag的字段，它就是来解决LW问题的。

比如，系统一定可以知道某个扇区具体属于哪个LUN，对于NAS，也一定会知道某个扇区具体属于哪个文件系统、目录ID、inode ID等，如果每次写入扇区的时候，都将对应的这些高层信息更新到DIF中的Application Tag并且一起写入扇区，那么如果今后一旦发生LW，当系统再次读出对应的扇区时，可能这个扇区原本对应的文件ID、LUN ID、inode号等是另外一个文件或者LUN的，而不是当前正在操作的文件或者LUN的，那么此时系统便会知道发生了LW，便会报错。但是绝大多数时候是不可能检测到LW的，因为基本上大部分写入动作都属于覆盖写，也就是某个扇区之前属于哪个文件，一般都会一直属于这个文件，对其更改之后，inode不会变化，发生了LW之后，其inode也依然不会变化；除非某个扇区对应的文件被删除，又有新文件占用了这个扇区，然后在发生第一次写入时，恰好遇到了LW，那么此时系统便可以通过两次inode不一致来判断发生了LW。而对于LUN来讲，基本上用这种方法就无法检测LW了，因为LUN一旦被创建，其位置基本上是固定的了，除非做了RoW的snapshot之类，既然总是固定，那么不管写入某个扇区多少次，都属于覆盖写，LUN ID都不变，也就无法检测到LW了。所以基本上目前能够检测LW的存储厂商屈指可数。

提示： LW对于一般存储系统来讲基本上属于必杀级别，一旦发生基本上所有产品都无法感知到，当然，它们一定不会告诉客户它们无法检测到LW，只能说这种事件发生几率非常低，

发生了的话你也就乖乖认了吧。但是有一家厂商却凭借其独特的数据排布架构而能够检测到LW，这家厂商就是NetApp。这里我绝对不带有崇拜或者推广的意思。

NetApp的WAFL文件系统，之前也介绍过，每次写入都会写到空闲空间，不会覆盖写，这种机制决定了它可以天然的检测到LW的发生。WAFL的文件系统Tree ID会随着每次刷盘而变化，只要系统读出某个扇区之后发现这个扇区中DIF区段Application Tag字段中包含的Tree ID并不等于上一次刷盘的Tree ID，那么就可以判断上一次刷盘时这个扇区发生了Lost Write，此时便可以通过RAID层面的Parity来重算出这个扇区原本应当写入的数据并做覆盖，其具体步骤与上文所述的相同。

（4）如何在SATA盘中存放DIF信息

SATA盘每扇区被低格为512B，那么就没有空余的空间来存放DIF信息了。此时就必须在上层强行分配DIF空间，有多种手段。如图18-54所示即为一种方式，也即是直接把磁盘当做一个连续地址空间，不管你一个扇区多大，我就在每512字节之后放8字节的DIF内容。这样做会导致520字节的逻辑扇区与512字节的物理扇区错位，导致每读出一个逻辑扇区，就要同时读出其横跨的左右两个物理扇区，写也一样，会产生大量的读写惩罚。

图18-54 SATA盘中的DIF放置方法1

另外一种方法则利用一种扩大粒度的思想。由于每512字节需要8字节的DIF，那么64个512字节，恰好总共需要 $64 \times 8B = 512B$ 的DIF内容，这恰好又是一个512B扇区的容量，那么就可以将这个专门存放DIF的扇区追加到每64个扇区之后。如图18-55所示即为这种思想的一个示意

图。

图18-55 SATA盘中的DIF放置方法2

但是这种做法有个问题，由于每次读出任何一个扇区，都需要顺带读出对应的DIF信息，所以如果仅仅读出比如第5个扇区，那么磁头此时就需要等待盘片继续旋转到DIF扇区位于磁头之下时，将其读出，增加了等待时间；并且，如果在设计DIF扇区所处的位置时，比如可以放在每64扇区的首部或者尾部，如果这种设计没有考虑盘片旋转方向与扇区排布方向的话，那么可能磁头会先划过DIF扇区，再划过其对应的64个数据扇区，那么此时磁头就只能在读取对应的数据扇区之后，等待盘片旋转一大圈，再转回到DIF扇区时，从而将其读出，所以这一点一定要考虑到，当划过DIF时就将其读出从而节省时间。其次如果遇到DIF扇区与其对应的64个数据扇区位于不同磁道的时候，还要额外产生寻道操作，这就更降低了性能。

经过实际测试，即便是考虑了时间先后问题，SATA盘此时所表现出来的性能确实也有不小的降低，就是因为等待时间与寻道开销导致。所以，有些厂家的设备不得不选择降低粒度，比如每8个扇区后面追加一个DIF扇区，这样的话， $8 \times 8B = 64B$ ，这个DIF扇区中就只有前64字节被利用了起来，浪费掉了448B，鉴于此，可以将这个DIF扇区后面紧邻的8个数据扇区的DIF信息也存放在其中，也就是一个DIF扇区的左右各8个共16个扇区的DIF可以共用这一个DIF扇区，这样就降低了浪费程度，变为384B的浪费值。同时提升了性能。可以调整为16扇区或者32扇区这样的粒度，从而在性能与容量浪费之间取得平衡。

（5）实现DIF计算和检查的角色及其所处位置

首先，阵列中的HBA上的FC、SAS控制器已经普遍支持DIF的计算、插入与剥离操作。当写数据IO被由主机下发到阵列前端HBA卡中的控制器芯片之后，芯片可以计算（计算出CRC，并插入对应的LBA地址）并向每个IO中插入DIF区段，之后将已经插入DIF的数据IO继续下发到IO路径下层。不仅前端，阵列后端连接磁盘扩展柜的HBA控制器也可以实现DIF的计算可插入。对于SATA磁盘，到底在阵列前端就实现还是在后端实现，取决于阵列设计者的全盘考虑。

其次，阵列操作系统内核软件模块也可以实现DIF的计算与插入，不过就需要耗费一定的主CPU资源了，但是如果要实现诸如LW监测这种级别的DIF，那么就一定需要在阵列系统内核中用软件实现了。

然后某些SATA-FC，SATA-SAS转接卡上的控制芯片也可以做到DIF计算与插入，这些芯片除了要计算DIF之外，还需要考虑SATA盘的512字节扇区对齐的问题，需要实现底层地址的屏蔽透明转换操作。有些SATA转接芯片则自己不计算DIF，只提供512字节到520字节的地址映射翻译工作，这样就可以让上层部件来计算和插入DIF了。

如图18-56所示为一个典型系统路径下DIF的插入、校验转发、写入、读出、删除动作流程示意图。可以看到阵列前端FC控制器实现了DIF区段的插入（写入阵列时）与删除（读出阵列时）操作，系统路径之下每个支持DIF的组件，比如后端HBA，都会对收到的IO数据进行校验，通过就继续向上层或者下层转发，一旦发现不一致则向其上游报错，从而上游可以重新发送或者连环向更上游组件报错从而推动问题的解决。

图18-56 实现DIF的角色与位置

另外，有些磁盘（比如日立）自身可以实现DIF区段的计算与插入，对上层透明为512字节扇区。在使用支持DIF的各种组件时，需要统筹掌握和利用，从而在路径上各个点处保持良好的配合。

3. 数据一致性保护总结

经过本节介绍，想必大家已经可以深入了解DIF属于一致性保护的方式，以及TW、MW和LW三种典型的使数据不一致的现象。由于主机侧的组件众多，所以目前业界也一直在研究如何保证端到端的数据一致性，让数据从被主机侧应用程序生成，一直到最后写入磁盘，都能够保证每个环节的一致性。

DIF依然不是万能药，其所能解决的一致性也是比较有限的。另外，XOR校验也并不能保证100%几率检测到不一致，比如一旦一串数据中的两个部位同时发生畸变，一处由0畸变为1，而另一处由1畸变为0，则此时这串数据的XOR校验值是相同的，系统不会认为数据不一致。ECC纠错算法则可以检测到这种不同位置的循环畸变。

另外，此系统发现某条带的数据存在不一致现象，但是每个扇区自身的校验都是一致的，那么此时系统就无法判断出到底是谁导致的不一致。而如果使用RAID 6双校验算法，就可以利用二元方程来判断究竟是哪个Segment发生了不一致，毕竟两个校验都发生不一致的几率是非常低的。

Lost Write，最令人头疼的难题，也不是不能解决，可以使用Write and Verify这个SCSI指令，即每次写入之后立即读出这个扇区来判断是否真的成功写入，但是这样做会耗费相当大的资源，不具实用价值，除非对数据一致性确有苛刻要求而又可以容忍性能降低的情况下。

现在业界出现了一种近线（Nearline，NL）SAS或者近线FC磁盘，这种磁盘其原型其实为那种带有SATA-FC或者SATA-SAS转接卡的SATA盘，由于磁盘阵列后端一般都使用FC Loop或者SAS Expander网络来连接FC或者SAS磁盘，而SATA盘若要用于这种磁盘扩展柜中，就需要加转接卡，各个厂商都去选择各种转接卡，导致型号、协议都不同，兼容性也不同，浪费资源。所以磁盘厂商干脆推出了将转接芯片集成到磁盘自身的SATA，这也就是所谓NL-SATA磁盘了。不仅如此，NL-SATA并不仅仅是这点变化，由于用于企业级存储系统中，所以厂商在低格的时候，对NL-SATA盘也直接低格成520字节扇区的格式以便实现DIF。

第19章 过关斩将——系统IO路径及优化

- LUN分布及虚拟化
- 系统IO路径图
- IO优化
- Cache
- OS内核IO流程
- 存储控制器IO处理过程

真正的存储系统性能优化是在研发机构的实验室中。要做到真正彻底的性能优化，需要从底层设计上彻底优化。目前大多数工程师所做的优化只不过是让底层已经设计好的架构充分发挥其应有的性能，严格来讲，这不叫优化，因为其本来就是应该被完成的。本章首先介绍系统IO路径上面的各个层次的固有的设计上的优化动作，然后会给出一些如何将这固有设计发挥出来，或者说如何不拖它们后腿的经验方法。

本章的精髓在于系统IO路径架构图。围绕这张图来论述图中各个模块和路径是怎样影响系统IO的。要想优化性能，必须胸有成图。作者精心制作了这张图，请读者务必牢记。

要理解存储系统，首先要理解计算机系统本身。存储系统中任何主体都是计算机，理解了计算机IO路径，也就理解了整个IT系统的IO路

径，只有这样才能目无全牛，胸有宏图，才谈得上最后去做优化分析工作，厚积薄发。

正像电影《黑客帝国》中所反映的那样，要真正地理解计算机都做了些什么，我们只有变身Nio去钻入其中一探究竟。但是目前做不到这一点，目前所做的是用程序来监测程序本身，比如debug程序。我们作为计算机世界的造物主，可能有时候也永远无法探究到被我们造出来的物种可能并没有按照我们规定的步骤去做。这也就是进化的开始。

本书前面的内容大部分都是游走于系统表层的一些功能表象，本章将刺破表层，到达存储系统的内部来一探究竟，看存储系统到底是怎么处理IO请求的。

如图19-1所示，本章开篇即给出了一幅系统IO路径架构图（传统基于Block协议访问的磁盘阵列），这幅图囊括了主机与存储系统与数据IO相关的全部主要逻辑模块与数据流的每个停站以及所有停站对数据所做的处理动作和功能模块。作者认为，要涉足性能优化方面，就必须在大脑中形成这样一张图，并且随手可以画出。在计算机中，一个IO请求始于应用程序，终于磁盘，在这条路径上，有形形色色的角色将对这个IO请求进行或转发，或地址映射，或排队，或拆分，或合并等动作。

图19-1 系统IO路径模块架构图

19.1 理解并记忆主机端IO路径架构图

总体来讲，系统IO路径所包括的主要角色模块就是上面列出来的这些，一个IO会依次经过每个角色，在物理上跨越主机与存储系统之间的网络，在应用程序和磁盘之间双向流动。本节论述IO的起源地也就是主机端的IO路径模块架构。

提示：本章的所有论述均基于机械硬盘，不考虑SSD。

19.1.1 应用程序层

应用程序是计算机系统内主动发起IO请求的大户，但计算机系统内可以向底层存储设备主动发起IO请求的角色不仅限于应用程序，其他处于操作系统内核层的模块，比如文件系统自身、卷管理层自身、适配器驱动层自身等，都可以主动发起IO。当然，只有应用程序发起的IO才可以修改用户实体数据的内容，其他角色发起的IO一般只是对数据进行移动/重分布/校验/压缩/加密等动作，并不会修改用户层面的实际数据内容。

应用程序在读写数据的时候一般是直接调用操作系统所提供的文件系统API来完成文件数据的读写等操作。有的应用程序可以直接调用卷管理层或者适配器驱动层API从而直接操控底层的卷或者LUN，比如一些数据库程序，它们直接操控卷而不需要使用文件系统提供的功能，它们自己来管理数据在底层卷上的分布。

应用程序发起的IO的类型是影响IO性能的最首要因素。本书第4章介绍了7对IO类型，每一对中包含的两个类型是相反关系。在此将这7对共14种IO类型做成了一个表格并且简要地列出了每种IO对系统整体相对

性能的影响。

其中，最优的IO方式为连续并发IO，如果追求吞吐量，则应当使用大块连续并发IO，如果追求IOPS，则应当使用小块连续并发IO。最差的IO方式为随机顺序IO，这种模式下不管是吞吐量还是IOPS都是相对很低的。比如某应用程序更改大量文件的文件名或者访问时间等属性，如果程序为单线程同步调用方式，则每次操作只能发起一次IO，每一轮只能更改一个文件，如果有几万个或者更多的文件，那么此时处理速度将会非常慢。解决办法是改为异步调用，或者多线程同步调用，让IO并发执行，充满IO Queue。

表19-1 IO属性对性能的影响

表19-1给出了各种IO形式对整体性能的影响，这些影响都是相对的，即总体来讲的普遍影响，不具有特殊情况下的意义。

每个应用程序都会有自己的Buffer用来存取有待处理的数据。应用程序向文件系统请求读数据之后，文件系统首先将对应的数据从底层卷或者磁盘读入文件系统自身Buffer，然后再将对应的数据复制到对应应用程序的Buffer中。应用程序也可以选择不使用系统内核缓存，这时FS将IO请求透明地翻译并转发给底层处理，返回的数据将直接由OS放到应用程序Buffer中。当应用程序向文件系统请求写入数据时，文件系统会先将应用程序Buffer中对应的数据复制到文件系统Buffer中，然后在适当的时刻将所有FS Buffer内的Dirty Page写入硬盘；同样，如果不使用系统内核缓存，则写入的数据经过FS文件—块地址翻译后直接由OS提交给FS下层处理。文件系统的动作是可控的，我们将在下一节描述文件系统的相关细节参数。

1. 同步调用和异步调用

应用程序可以选择两种IO执行方式：第一种是同步IO调用，另一种是异步IO调用。

如图19-2所示，前者指某线程向OS发起IO请求之后便一直处于等待挂起状态，直到OS将IO结果返回给这个线程；后者指某线程向OS发起IO请求之后，OS立即返回一个“已接受”信号，线程此时就可以继续执行后续代码（后续代码可以执行对以前接收到的数据的分析处理或者继续对OS发起后续的IO请求），当OS将IO数据从相应介质读入或者写入完成后，会利用某种方式让线程知道（比如Windows系统异步IO常用的IO Completion Port机制）。有人可能会在此产生疑问，如果遇到某个程序必须等待当前IO数据返回，不返回的话就无事可做而挂起呢？这种情况是有的，如果真的遇到这种情况，只能说明存储系统或者数据链路的速度太慢已经影响程序性能了，但是这里请注意，如果使用了异步IO调用，则可以在短时间内发送大批IO请求到OS，从而接收到大批量数据返回，如果这时应用程序依然没被喂饱，那么说明存储系统性能无法跟上；而如果程序虽然使用了异步IO调用但是却与同步IO时做相同的动作，即一个IO请求发出后，程序虽然没有被阻塞但是依然自我等待，一个IO一个IO的来，那么实际也是没有任何效果的。

图19-2 同步异步IO示意图

前文中所述的并发IO，其在应用层（并发IO在每个层次都有对应的意义）的意义就是指异步IO模式，批量发起IO从而喂饱底层存储系统，榨取极限性能而又不能让其过载。过载的后果就是IO响应时间急剧增加而IOPS和带宽吞吐量根本没怎么增加。

IO请求发送到OS内核之后到内核将IO请求对应的数据读取或者写入完成这段时间会贡献为OS内的IOWait指标增高，IOWait指标一旦升高到高于60%左右的百分比，那么就需要考虑后端存储系统所提供的性能是否已经不能满足条件了，需要检查各项配置以确认是否是存储系统本身能力有限还是配置没有最优化。

同步IO调用如果不加任何参数的话一般是操作系统提供的默认调用方式，也是一般应用程序首选的IO调用方式。一般情况下，如果遇到数据链路速度或者存储介质速度很慢的情况，比如通过低速网络进行IO（Ethernet上的NFS、CIFS等），或者使用低速Flash芯片等时，使用异步IO方式是一个很好的选择。第一是因为IO请求发出后线程可立即执行后续功能，比如继续处理已接收到的数据等；第二是因为异步IO调用可以接连发出多个IO请求，一同发向目标，目标在接收到这些IO请求之后可以一并处理，增加效率。比如SATA硬盘的NCQ功能，如果应用层只有一个线程，而且整个IO链路上都是同步调用的，那么这个Queue中永远就只有一个IO请求，想排队优化都不可能，磁头臂只能按照IO顺序寻道。

要实现与多线程异步IO类似的结果，可以采用另一种方法，即生成多个线程或进程，每个线程或进程各自进行同步IO调用。然而，维护多个线程或进程需要耗费更多的系统资源，而采用多线程异步IO调用虽然需要更复杂的代码来实现，其相比多线程同步IO的方式来说仍然更加高效。

注意：在Windows系统中，如果应用程序在打开某文件进行读写操作时未指定特殊参数，则文件系统默认是使用自身缓存来加速数据读写操作的。并且，这种情况下异步调用多数情况下会自动变为同步调用，结果就是IO发出后操作系统不会返回任

何消息直到IO完成为止，这段时间内线程处于挂起状态。为何会这样呢？有三个原因。

（1）预读和Write Back。文件系统缓存的预读机制可以增加IO读操作的命中率，尤其是小块连续IO操作，命中率几乎百分之百。在这种情况下，每个读IO操作的响应时间会在微秒级别，所以OS会自动将异步调用变为同步调用以便节约异步IO所带来的系统开销。

（2）尽量保持IO顺序。异步IO模式下，应用程序可以在单位时间内发出若干IO请求而等待OS批量返回结果。OS对于异步IO结果的返回顺序可能与IO请求所发送的顺序不同，在不使用文件系统缓存的情况下，OS不能缓存底层返回的IO结果以便重新对结果进行排序，只能按照底层返回的实际顺序来将数据返回给应用程序，而底层设备比如磁盘在执行IO的时候不一定严格按照顺序执行，因为文件系统之下还有多处缓存，IO在这里可能会被重拍或者有些IO命中，有些则必须到存储介质中读取，命中的IO不一定是先被发送的IO。而应用程序在打开文件的时候如果没有给出特殊参数，则默认行为是使用文件系统缓存的，此时系统内核缓存便会严格保持IO结果被顺序地返回给应用程序，异步调用变为同步模式。

（3）系统内核缓存机制和处理容量决定。文件系统一般使用Memory Mapping的方式来进行IO操作，将映射到缓存中一定数量的page中，目标文件当需要的数据实际内容没有位于对应page中时，便会产生Page Fault，需要将数据从底层介质读入内存，这个过程OS自身会强行使用同步IO模式向下层存储发起IO。而OS内存在一个专门负责处理Page Fault情况的Worker线程池，当多个应用程序单位时间使用异步IO向OS发送大量请求时，一开始OS还可以应付，接收一批IO对其进行异步处理，但是随着IO大量到来，缓存命中率逐渐降低，越来越多的

Page Fault将会发生，诸多的Worker线程将会参与处理Page Fault，线程池将迅速耗尽。此时OS只能将随后到来的IO变为同步操作，不再给其回应直到有Worker线程空闲为止。如果应用程序在打开文件进行操作时明确要求不使用文件系统缓存，那么就不会受上述情况的制约，OS会直接返回“接收”信号给应用程序同时将请求发送到文件系统做文件—块地址翻译后转发到下层处理。

导致Windows将异步强行变为同步的原因不只有系统内核缓存的原因，其他一些原因也可以导致其发生。在Windows系统中，访问NTFS自身压缩文件、访问NTFS自身加密文件、任何扩展文件长度的操作都会导致异步变同步操作。要实现真正的异步IO效果，最好在打开文件时给出相关参数，不使用内核文件系统缓存。但是不使用文件系统缓存会引起读IO请求过慢，采用越过缓存并且使用异步IO的应用程序都是经过严格优化的，它们自身会实现预读操作从而将数据预先读入自身的缓存。这种不使用内核文件系统缓存的IO方式又被普遍称为“Direct IO”或者“DIO”。异步IO模式又被称为“AIO”即“Asynchronous IO”。

2. Windows系统下的Asynchronous Explorer

在这里向大家介绍一个工具：Asynchronous Explorer。这个工具为一位国外高手开发，他为了研究异步IO的行为而专门开发了这个工具。它可以让你直截了当地判断在执行IO操作时Windows系统底层都做了什么事情，发生了什么，是否某处有缓存在发生作用等。有兴趣的读者可以在作者的博客留下E-mail以便作者将这个工具发送过去。

如图19-3所示为Asynchronous Explorer的主界面。程序首先需要在磁盘上任意目录生成一个结构化的文件，这个文件中存放着大小为1KB的若干条记录，初始时需要创建这个文件，指定其包含记录的条数。创

建完后，在窗口中部区域填入测试将要发送的IO请求数量，每个请求会读出文件中的一条记录，也就是每个读IO请求为1KB（两个扇区）大小。然后单击Generate按钮生成测试需要发送的IO列表，生成的IO完全是随机IO。然后单击Run按钮，程序便会以异步IO模式向OS发起请求，所有IO将尽量一次性发出。

图19-3 Asynchronous Explorer主界面

窗口中部给出了可选的参数，Scan Suggestions在下一节介绍。其中有个调用参数比较重要，即FILE_FLAG_NO_BUFFERING，勾选这个参数表示应用程序在打开文件进行IO的时候不使用文件系统缓存（读和写IO都不进入系统内核缓存而直接进入卷或者设备驱动IO层面），FILE_FLAG_WRITE_THROUGH在下一节介绍。

窗口下部为本次测试的结果标示图，每一个黑点表示一个IO，对角线为同步线，即如果某个黑点落在了这条线上，那么就表示这个IO的结果是被按照请求发出的顺序返回给应用程序的，如果黑点落在同步线之上，则证明IO结果早于被发出的顺序被返回，黑点落在同步线下方则表示这个IO结果晚于被发出的顺序返回。图中有波动幅度的曲线为IO结果返回的时间与IO地址偏移量的函数曲线。

现在我们来分析一下图19-3中的曲线和IO分布情况并做出几个判断。测试时由于选择了FILE_FLAG_NO_BUFFERING，所以本次测试为纯异步IO模式，所有IO在最短的时间内批量发送到了OS内核。然而OS返回的结果却不是按照IO被发出时的顺序返回的，有早有晚，基本上平均分布，这种现象的原因是因为FS之下某处有某种Queue在发生作用，典型的例子就是硬盘中的Queue，比如SATA盘的NCQ。Queue会将IO按照当前磁头臂所在的位置按照电梯原理进行优化重排，使得磁头臂

降低寻道次数和寻道跨度从而优化性能。这一点可以从IO偏移量曲线判断，可以看到IO结果基本上是按照偏移量递增的顺序返回的，而IO发送的时候却是完全随机的，所以这里判断底层的某个Queue对随机的IO按照偏移量做了重排，使得磁头臂顺序地并且小跨度地寻道依次完成了所有IO的读取动作并且返回结果。偏移量曲线产生了4次平滑波动，第一次向下，第二次长时间平缓向上，第三次陡峭向下，第四次陡峭向上，可以判断这些波动对应了磁头臂的摆动方向，磁头臂做了四次大幅度摆动，而每次摆动周期内伴随着多次小跨度换道操作，这种磁头臂寻道方式是很优秀的。

在主界面中选择Processing Time Plot单选按钮可以切换到另一种结果显示模式，即IO发送顺序与对应IO响应时间及IO偏移量的曲线图，如图19-4所示。上部曲线为IO响应时间，范围跨度比较大的曲线仍然表示IO偏移量曲线。可以看到IO偏移量是来回摆动非常随机的。由于Bypass掉了文件系统缓存，所以IO响应时间的数量级上升到了100ms的级别。

图19-4 响应时间和偏移量曲线

我们再来看一下使用系统内核缓存之后的IO变化情况，如图19-5所示。虽然程序依然使用异步IO调用，但是此时OS已经强制变为同步IO操作并且保持了IO结果返回的顺序符合发送时的顺序。可以看到每个IO都落在了同步线上，两个模式显示的偏移量曲线相同，表示IO结果返回的顺序与发送时的顺序相同，并且由于系统内核缓存预读的影响，测试文件大部分已经处于缓存中，所以IO响应时间下降到了100微秒级别，降低了1000倍。

图19-5 使用系统内核缓存之后的IO变化

进行完上面的测试之后，FS下层某处的Buffer（比如磁盘自身缓存）一定残留有之前预读的部分数据。为了验证这一点，我们即刻勾选FILE_FLAG_NO_BUFFERING，并且不再重新Generate新的IO列表，使用原有的列表以保证缓存命中率。立即Run起来。结果验证了这个说法。

如图19-6所示，IO发起之初一段时间内结果返回非常迅速，延迟在100微秒级别，全部落在了同步线上。既然没有使用系统内核缓存，这么快的响应速度是从哪里来的呢？答案只有一个，即FS下层某处必定有一处或者多处缓存在发生作用，这个或者这些缓存内残留有之前程序读取过的数据部分，所以开始的一部分IO命中，直接从缓存返回给了应用程序，而且可以发现偏移量曲线随机跳跃，表明IO结果是按照发送顺序迅速被返回的。而随后的IO命中率变得越来越低下，最终成发散形状分布，这时底层的Queue优化效用又被显现出来，偏移量曲线变得平滑，响应时间也相对上升到了10ms级别。

图19-6 验证FS下层某处缓存的作用效果

测试时勾选了一个选项，即显示IO发送和IO返回的追逐条，如图19-7所示。上面的条表示IO被发送出去的完成量，下面的条显示IO结果返回的完成量，缓存命中时两个条的行进速度是相同的，而缓存不命中时IO完成量进度条就会落后于IO发送进度条。本测试样例中，起初一段时间两个条行进速度相同，随后IO完成量条逐渐变慢，对应了测试结果。如果降低测试读取的数据量，比如读取1×100条记录，那么100KB的数据对于磁盘缓存来讲是个小数目，被完全留在磁盘缓存的几率会增加，此时再次测试就会发现偶尔会出现IO完全落在同步线上的情况，虽然Bypass了系统内核缓存。

图19-7 异步IO 的追逐条

下面，我们用Asyn Explore来验证一下在纯异步IO模式下，应用程序的IO是否是批量发送的。我们使用iSCSI协议来连接一个外部存储系统中的LUN，并在主机端利用Wireshark工具来抓包分析。如图19-8所示，在5号TCP包中竟然一次性包含了30个SCSI Read请求IO，可见异步IO确实是在短时间内批量被发出的，以至于30个IO一次性被打包在了一个TCP包中传送。后面的几个加长的包同样是这种情况，当然也夹杂着只包含一个IO请求的TCP包。对IO的Response结果会在随后一个一个地被返回。另外，从每个IO请求的长度=2来看，确实符合Async Explore每次读IO请求1KB数据相符（ $2 \times 512\text{B} = 1\text{KB}$ ）。

图19-8 验证纯异步IO的批量发送

注意： 在使用ISCSI/NAS等基于TCPIP的存储协议时，并且是随机IO环境下，利用Bypass文件系统缓存的纯异步IO模式会大大减少网络传输的开销。因为如果使用文件系统缓存，则OS强制变为同步IO模式，IO一个一个地执行，所以每个IO请求就需要被封装到单独的TCP包和以太网帧中，在IO量巨大的时候，这种浪费是非常惊人的。图19.8所示的例子中，纯异步IO模式下的抓包数据文件只有120KB，而如果使用FS缓存之后，同样的IO列表，抓包数据变为490KB，开销是前者的4倍，这是个绝对不容忽视的地方。

注意： 不管Windows还是Linux，都是“用户程序—OS内核—存储设备”这种架构，用户程序和OS内核之间存在一套IO接口。同样，OS内核与存储设备之间一样存在着IO接口，也有异步同步、Write Through、Write Back等参数。Linux操作系统的IO行为与Windows有很多不同之处。

注意： 在Windows系统下，OS内核与存储设备之间的IO行为完全与用户程序与OS内核之间的IO行为对应，即上层为同步调用则底层也同步，上层为异步调用则底层也异步（在不使用FS缓存时）。

而在Linux系统下，上层的IO与底层不一定是对应关系，比如，如果使用文件系统进行IO并且Mount之后，底层对磁盘的操作会严格按照Mount时给出的选项（Async或者Sync）来执行，并且，如果Async Mount而同时用户程序进行同步调用，则底层依然是异步操作。对于不使用文件系统进行IO的情况，比如使用dd程序直接操作设备层，由于dd采用同步调用，而底层卷设备不可能有预读机制，所以对于dd的同步读操作，在底层也反映为对磁盘设备的同步读操作；但是对于Write Back模式的写操作，由于底层卷处存在缓存，这个缓存会在接收写操作之后即返回成功给上层，然后对IO进行优化（IO Combination等），然后批量异步Flush到物理存储设备。卷处的缓存不但可以Write Back，而且还拥有读缓存作用（虽然它不可进行预读操作，但是之前被读入数据会残留一定时间），如果上层在短时间内发起重复地址的读操作，会先从卷缓存处寻求命中，不命中后再读取磁盘。

注意： 这里还需要理解一点，即如果使用文件系统缓存，那么Windows会强行将异步IO变为同步阻塞模式，即应用无法在短时间内发送大批IO了。这样做虽然使用了文件系统缓存的加速作用，但却也降低了IO并发度。而如果在FS下位某处的缓存同样具有加速IO的作用，那么不使用FS的缓存从而让应用层处可以产生并发的无阻塞异步IO，而这些IO命中于FS下位的某处缓存，这样就可以一箭双雕了。本例中由于测试只是用了台式机单块磁盘，磁盘缓存容量小且处理能力有限，所以IO响应

很快升高。如果底层设备为具有大容量缓存和高处理能力的智能存储设备，那么缓存命中率会大大增加，这种情况下就会获得更快的响应时间，是最优的实现方式。如图19-9所示，测试使用一台外部NAS存储系统，具有大容量缓存和高处理能力，即使使用了FILE_FLAG_NO_BUFFERING参数，依然得到了比较高的响应速度和向同步线归拢趋势的IO分布。分别利用CIFS和iSCSI作为访问协议，得到了相同的结果。左面是CIFS的结果，右面是iSCSI的结果。iSCSI环境下读取了1000条记录，所以IO分布点比CIFS多，后者读取了100条记录。

图19-9 FS下层有缓存的效果

注意几种表述方式：异步IO不一定不使用缓存，同步IO不一定使用缓存。同步异步和Write Back/Through没有直接关系。

3. Network IO

有时候文件并不存放在本地磁盘，而是存放于另外一台计算机上，需要通过某种网络比如TCP/IP来访问对方的计算机，并且还需要某种网络文件访问协议比如CIFS/NFS等来对位于网络另一端的计算机上的文件进行IO操作，这便是Network IO，即网络IO。

如图19-10所示为网络IO的示意图。其中有一个关键模块为Network Redirector。Redirector的作用是将上层的IO请求翻译成网络文件访问协议后发送到对端，这样，应用程序就可以像访问本地文件一样通过网络来访问其他计算机上的文件了。

图19-10 Network IO流程

Network IO的过程如下。

应用层向OS发起IO请求读写位于某网络映射盘符下的某个文件。

OS接收到这个IO请求之后，交给Network Redirector处理。Redirector接收到IO请求之后，将这个IO翻译并封装成对应网络协议与网络文件访问协议相匹配的数据包并将其发送到对应的远程计算机。

远程计算机上的Network Redirector接收到这个IO请求之后，对源计算机进行认证，认证通过之后，远程计算机根据IO请求中的具体信息对本地的目标文件进行IO操作，然后将数据通过Redirector返回给源计算机。

源计算机收到远程计算机的数据之后，将数据返回给发起IO的应用程序。

注意：网络IO的响应时间普遍较长，有时甚至会出现超时的情况，应用程序必须做好准备，延长IO等待超时时间。

由于外部网络稳定性不比内部总线，所以会经常发生中断现象从而导致TCP连接中断而网络文件访问协议栈也会Reset。此时Client端应用程序并不会感知这一点，可能依然会使用之前对文件进行打开操作时所获得的文件句柄对文件进行操作，而Server端由于协议栈Reset导致句柄失效，所以会返回句柄过期的通知给客户端Redirector，Redirector便向上层报错，应用程序此时需要重新进行打开文件操作以获得新句柄。这种现象在使用NFS协议时非常常见。

4. Network IO下的缓存机制

由于网络IO速度远比不上本地IO，所以如果在Client端保持一个缓存的话是非常有效果的。网络IO的Client端缓存可以同时加速读和写IO。NFS和CIFS都可以使用缓存。

CIFS下的缓存和IO机制

下面介绍Buffered IO和非Buffered IO模式下的读行为。

使用系统缓存是程序在Open一个文件时默认的选项，不管对于CIFS还是本地文件。在此，很多程序员不会去过多考虑这一点，但恰恰就是这一步有失考虑，在特定情况下却可能造成程序IO性能极差。我们来做一个疯狂实验。

使用Async Explorer在一个CIFS协议映射的网络驱动器中生成一个测试用文件。然后生成一份测试地址列表，勾选FILE_FLAG_NO_BUFFERING不使用系统内核缓存，异步地随机读取文件中的100条记录（每条为1KB大小），开启Wireshark工具抓取网络上的CIFS协议数据流，然后单击Run按钮，得到的结果如图19-11所示（只列出了部分结果）。

图19-11 不使用缓存异步IO

我们从结果中可以看到，在Bypass系统缓存后，Network IO模块完全透传了应用层的IO请求，IO类型确实是只有异步模式才可能让单线程发出的并发IO，而且每个IO大小都是1KB。当然第一个IO为0.5KB并且是同步的，这有可能是程序故意而为，比如读取文件标识等信息。可以看到35~40号包表示了接连发出的6个并发IO请求。而且每个IO针对的文件Offset也都是随机的。

我们现在再使用同样的IO地址列表，针对相同的文件，做相同的测试，唯一区别就是不勾选FILE_FLAG_NO_BUFFERING，也就是使用系统缓存。测试结果如图19-12所示（只列出了部分结果）。

图19-12 使用缓存，读惩罚

从以上结果看来，OS确实将异步操作强制变为了同步操作，除此之外，还有一点最大的不同，即IO Size变为了4KB或者8KB了。但是程序明明给出的是1KB的IO请求，为何到了底层反而被扩大了如此之多呢？考虑到4KB和8KB这种数值，我们首先应该想到操作系统的Memory Page大小为4KB，这里是不是OS使用了Memory Mapping方式将IO的目标映射到页面内存地址空间了，从而针对这些目标的读写都会以Page为单位呢？要调查这一点，我们就要首先假设这个结论成立，而如果结论成立，那么OS在读取对应Page的时候一定是从Page的边界开始读的，也就是说，每个IO请求的Offset值必定可以除4KB而除尽。我们可以算一算，上图中的每个Offset都是可以除尽4KB的，所以，这就充分证明了这个结论。另外，由于两次测试需要读取的文件偏移地址列表是相同的，但是第二次测试结果中每个IO的Offset却并不等于第一次测试结果中对应的值（高位大致相同），这也说明了，第二次测试，Network IO模块并没有透传上层的IO请求到CIFS Server上，而就是走入了Page Fault缺页处理流程。

细心的读者可能已经发现，测试1中的37号包对应的Offset为2560的IO请求，在测试2的结果中不见了，并没有对应的IO被发送到CIFS Server。这是怎么回事呢？我们算一算，这个请求的Offset是从第2560B开始的后1024B， $2560 + 1024 = 3584\text{B}$ ，而文件被映射到内存的第一个Page大小是4096B， $3584 < 4096$ ，落入Page范围内，那么OS应该读入这个Page的啊，为何第二次测试结果并没有读入呢？我们立即可以推断，

由于第一个IO的Offset是0，长度是512B，同样也落入第一个Page，是不是OS会对已经读入的Page做一定时间的缓存呢？测试结果充分证明了这个行为，正因为37号包对应的第4个IO请求发生了Read Cache Hit，所以第二次测试时Network IO模块根本没有将这个IO请求发送给CIFS Server。

那么为何会出现8KB尺寸的IO呢？要解释这个问题很简单，我们依然将两次测试结果来进行比较。在第一次测试结果中找到与二次结果中的63号包相对应的包，即42号包，一次结果中42号包对应的IO请求为：从2334208开始的1024字节， $2334208 + 1024 = 2335232B$ ，而2334208～2335232这个Offset范围会落入哪个Page呢？大家可以计算一下，结果是前一部分落入一个Page，后一部分落入下一个Page，这比较尴尬，所以OS会读出这两个Page，也就是共8KB的内容来应对这个IO请求。

还可以从测试结果发现，即使Open文件的时候，程序选择了使用缓存，客户端文件系统也丝毫不会做任何程度哪怕是最小力度的试探性预读操作，但是却可以缓存已经被读入的Page。

在客户端使用Filemon监测工具

图19-13 Filemon 监测结果

同样监测到了相同的结果，在另一个测试中，出现了同样的结果，如图19-13所示。带星号的表示非程序发起的IO请求，说明是由OS内核自己发起的IO请求。其中IRP_MJ_READ是内核IOM提供的功能函数，详见本章后面部分。

提示：
FILE_FLAG_NO_BUFFERING或者
FILE_FLAG_WRITE_THROUGH这两个参数对于写IO请求来讲具有大致类似的效果，但是对于读IO来讲，想越过FS缓存就只能使用前者。如果目标文件位于网络另一端，那么

FILE_FLAG_NO_BUFFERING这个参数只会对客户端发生效果，当请求被传送到服务器端时，对于读请求，这个参数不会产生任何效果，服务器总是可以首先进行目标文件在其本地缓存的Cache Hit流程；但是对写请求会有效果，服务端在SMB协议中会检测这个参数对应的标记以便做出相应动作，设置了标记的请求，服务端会保证将其写入硬盘之后才返回完成信号。如图19-14所示为只设置了前者（上半图）和只设置了后者（下半图）时数据包中不同的位被置1。另外，这两个参数是有本质上的区别的，详见下文。

另外在其中我们还可以看到其他本章前文所描述的一些FLAG，比如控制随机或连续IO的FLAG。

图19-14 Write Through和No Buffer位

至此我们得出了结论：对于小块随机IO读，使用CIFS方式访问数据的情况下，客户端在调用API时使用缓存的两个坏处是：异步变同步、读惩罚。

说到写惩罚大家很熟悉，但是读惩罚确实也存在。所以我们要尽力避免读惩罚，由于开发人员在设计程序时如果没有显式地认识到这个问题，一般都会直接使用默认参数，没有越过系统内核的缓存，而这恰恰就是导致性能低下的原因。而对于大块连续IO读，更没有必要使用客户端缓存，因为内核在CIFS的IO方式下不会预读，还要它做什么甚呢？

（实验证明，即使使用Async Explorer在打开文件时指定FILE_FLAG_SEQUENTIAL_SCAN参数来刺激FS预读也是无济于事，试验结果就不再引入了）使用客户端缓存唯一能够有加速效果的情况，是小块连续IO读，比如512B的IO。这样，客户端每读入1个Page，就相当于预读了后面7个IO所需要的数据。称这种行为为预读尚且有些自欺

欺人，况且连续小块读这种行为在应用层就应该尽量将这些连续的小块合并为大块才是正路，不应该不负责任地增加底层的负担。

曾经有人在Windows Vista 64B版本上遇到了同样的问题，而且读惩罚更加严重，每次需要读入32KB的数据，但是这里作者不能确认是否是Windows Vista 64B Edition的Page Size确实是32KB。所能确认的是，上文测试中的4KB确实是Page Size在起作用而不是FS格式化时指定的Block Size在起作用。退一步讲，CIFS协议本身并没有引入Server端向Client端通告Server本地FS Block Size的机制，所以客户端既然不知道Server的FS Block大小，那么上文测试中的4KB也就与FS Block Size无关。为了证明这一点，下面做一个试验给大家看一看。

如图19-15所示，F盘是一个本地磁盘，被格式化为FS Block Size=2048KB。图中第一行表示程序向F盘下面的一个文件发起了Offset=19949，Length=1024B的随机Offset写IO，这一定会产生写惩罚。果然，第二行带星号，表示这是内核自动发起的IO，是一个读请求，可以看到这个读请求的Offset=16384，Length=4096B。其Offset可以被4096除尽，表示OS自动读入了1个Page大小的数据，Page在内存中都是以4KB为单位的，所以这个IO的起始Offset也是出于Page的边界上。虽然FS被格式化为2048B为一个Block，但是这里OS依然使用了4KB的IO Size，这就充分说明了上文的结论。

图19-15 Filemon写惩罚监测结果

另外，根据这个图我们还可以看得出来，OS读入这个Page之后，第一行的IO会落入这个Page，图中所显示的所有IO请求也都落入了这个Page，所以OS再没有发起其他Page的读IO请求，因为图中所有的IO都会Cache Hit。当然这里只是特殊情况，因为可以看到图中的所有IO的起始Offset根本没相差几个字节，测试用程序也是故意定制成这样的。如

果遇到一个起始Offset没有落入已读入Page范围的IO请求，那么内核依然还要读取对应的Page。本章后面章节讲述的块设备IO的IO Size等也都会受Page Size的影响。

对于小块连续读，使用Buffered IO会有一定的缓存命中率，但是由于底层没有主动预读，所以带来的提速是微乎其微的。

提示： FS Block Size大部分情况下并不决定FS预读或者写惩罚读的单位，FS往往是以内存页面的大小来决定读入数据单位的，包括下面章节要讲的AIX系统下FS预读，也是使用Page来作为单位的。FS Block Size的最大一个作用其实就是单纯地作为FS对文件空间进行分配时的最小单位。所以，这里请准确理解OS RAM Page Size和FS Block Size的角色和作用。

上面论述了CIFS环境下的读行为，现在我们再来看看写行为。对于写IO，如果程序明确指出了使用FILE_FLAG_NO_BUFFERING或者FILE_FLAG_WRITE_THROUGH的话，那么底层会完全透传上层程序发出的写IO，并且这个请求到了CIFS Server端也会被Server以Write Through的方式写入磁盘后才会返回完成信号。而如果程序没有使用这两个参数的任何一个，那么客户端系统会默认使用缓存进行写IO。

我们首先来调查系统对非Buffered IO的写行为。非Buffered IO又分为WriteThrough方式和NO_BUFFER方式，上文小提示中给出了二者的本质区别，这里我们用实践来检验之。

非Buffered IO（Direct IO）模式下的写行为如下。

下面我们分析在CIFS访问方式下，使用FILE_FLAG_NO_BUFFERING参数打开目标文件之后，系统内核是否

还会发生惩罚行为。

如图19-16所示，使用测试软件以相同的其他测试参数，分别用WT模式和FILE_FLAG_NO_BUFFERING方式打开目标文件进行513B（注意，非512B对齐）IO Size的随机读测试。

图19-16 WT模式与NOBUFFER模式读对比测试

图中右侧下部1号记录显示了在使用FILE_FLAG_NO_BUFFERING参数打开文件后，底层所监测到的NoBuffer Access。下部左侧的CIFS协议数据包中对应的FLAG“Intermediate Bufferding”也置了1，两者一一对应。两次测试给出的内核IO惩罚完全不同：WT模式下惩罚严重，而NoBuffer模式下根本没有惩罚。而且还得出了另外一个结论，即如果底层采用CIFS协议来访问目标文件，则上层程序在使用FILE_FLAG_NO_BUFFERING之后，可以发出iosize不等于扇区整数倍的IO请求，而且这些请求被完全地透传到CIFS层执行。这个设计是合理的，因为CIFS协议自身并没有规定读写请求的offset必须与某个单位对齐。而对于底层直接使用物理磁盘IO的情况，就必须要求使用NoBuffer模式的程序发出扇区对齐的IO请求了。

使用了NoBuffer模式之后，针对CIFS目标文件的读和写请求均被完全透传。如图19-17所示为同样使用了NoBuffer模式对目标文件进行567B的随机写入，发现IO请求被完全透传到底层，没有任何读或者写惩罚存在。

图19-17 NoBuffer模式写测试

提示： CIFS访问方式下，使用NoBuffer模式打开文件进行IO的程序没有必要一定发出扇区对齐的IO请求，此时内核完全透传上层的IO请求。但是IO请求到了CIFS Server端之后，依然会在Server端本地产生对应的惩罚，所以，从全局角度来讲，程序任何情况下都发出扇区对齐的IO请求是最理想的状态。

5. Buffered IO模式下的写行为

这里需要区分概念，Buffered IO一样可以使用Write Through（WT）。WT并不是说IO请求不经过内核缓存，而是先进入缓存但是系统需要将数据彻底写入底层介质之后才返回成功。

对一个文件进行写入操作有两种方式：一种是追加写，另一种是覆盖写。这两种模式对系统底层的IO行为影响迥异。下面会分别介绍。

1) Buffered IO下使用WRITE THROUGH模式写

首先我们对WT模式做一系列疯狂实验。在Windows系统下使用某测试工具打开CIFS Server上某文件进行测试，打开时只指定WRITE THROUGH参数，然后从文件头部开始做1KB每次的同步写操作。从如图19-18所示的底层IO输出结果来看，产生了严重的读和写惩罚。

图19-18 WT模式连续小块写测试

图中左侧为Client与Server之间的IO请求输出，右侧为Client本地使用Filemon底层过滤驱动检测到的本地所发生的IO请求输出，发现二者是一一对应关系。

为了满足第一个offset=0 length=1KB的写操作，系统首先需要读入4KB大小的Page0也就是offset=0 length=4KB的数据（对应右侧带星号的75号记录），然后将读入的page的前1KB在内存中替换为IO请求的数据。由于设置了WT参数，所以此时系统还必须再次将此修改后的page0写入，所以紧接着发生了4KB的写入请求（对应右侧带星号的76号记录），第一次1KB写IO至此完成。第二次程序继续发起写IO，由于是连续地址写，这次IO为offset=1024，length=1024。但是这次，系统不再读入page0，因为前一轮修改后的Page0依然在缓存中，此时系统直接使用这个page0即可，将page0中从offset1024开始之后的1024B替换为本次IO对应的数据，然后立即发起向CIFS Server的写入请求，写入修改后的page0。对于再往后的2次写入，系统依然不需要从Server端读入page0，所以，对一开始发生的4个写IO请求，系统只读入了一次page0，却写入了4次修改后的page0。依此类推，对下4次写IO，系统需要读入page1，然后执行相同过程。

总体来讲使用WT模式，并没有完全Bypass系统内核的缓存，所以直接导致了严重的读写惩罚，产生了4倍的总惩罚量，计算公式为（总读入+总写入-应写入）/应写入。图中右侧1号记录中可以看到过滤驱动监测到了上层打开文件时指定的WT参数，带星号的记录表示这个请求是内核发起的而不是程序发起的。

同理，如果将上面的测试程序发起的IO Size改为2KB，那么推导可得出每2次上层IO就会对应内核的1次读page和2次写page动作，图19-19给出了结果，惩罚倍数为2倍。同理可得，如果上层IO Size为512B，那么惩罚倍数将达到8倍。

图19-19 WT模式2KB写入测试

同理还可得，如果程序发起的IO Size为4KB，那么惩罚倍数=1，也就是没有惩罚，如图19-20所示。如果程序发起page size也就是4KB对齐的IO请求，系统底层是否还会产生惩罚？可以看到系统底层并未出现任何惩罚量。右侧的监测记录也表示程序发起的IO写与内核发起的IO写一一对应。

图19-20 WT模式4KB对齐IO测试

上面的测试用例全部是连续地址写入，从结果也可以判断，连续地址写入模式下，由于会有部分IO缓存命中，所以一定程度上降低了惩罚倍数，惩罚倍数=pagesize/iosize，最小为1。下面我们设计另外的用例来避免缓存命中，看一看此时惩罚倍数的变化。如图19-21所示为WT模式下随机1KB写入行为测试结果。可以看到，随机模式下由于缓存命中率几乎为0，所以每次程序发起的IO都会导致内核产生1个page的读和1个page的写操作，1KB的惩罚倍数为7倍，同理可得512B的惩罚倍数为15倍。

图19-21 WT模式随机1KB写测试

2) 追加写底层IO行为

在Windows系统中使用测试工具从0开始追加写一个文件，每次写1KB，同步+连续+BufferedIO模式。结果如图19-22所示。

图19-22 以1KB为单位追加写

在图19-22中，左侧为在操作系统底层过滤驱动监测到的本地IO行

为，右侧则为CIFS服务端检测到的实际接收到的客户端IO。在左侧的窗口中我们可以看到，程序确实以1KB为单位不断写入文件。这里无法看出是追加还是覆盖，但是我们可以确定就是追加，因为程序是以这种模式执行的。由于发生的IO过多，图片中作者人为地删掉了1000多行记录，最终程序从0写入了共 $523264 + 1024 = 524288\text{B}$ 。随后，程序发起了一个FLUSH_BUFFER操作，通知操作系统将方才写入的数据Flush到存储介质，当然这里的存储介质其实是CIFS服务端。所以我们看到操作系统将缓存的数据写了出去，对应左侧的带星号的记录，同时对应了右侧的632号之前的包。可以看到客户端是以64KB为单位写入CIFS Server的。随后，系统内核发起了一个Cleanup操作，这个操作被透传到了CIFS Server执行，所以在右侧可以看到Server接收到了Flush Request，并在数据成功写入Server本地磁盘之后，返回Response。

综上所述，Buffered IO模式下的追加写操作，是不会引发任何写惩罚的，其原因是因为待写入的部分原来并不存在，文件长度是逐渐增加的，所以客户端内核不可能读入一个并不存在的Page映射块。追加写模式下，内核会将写IO数据暂存并做合并操作，合并为更大的IO，提升了效率，所以追加写使用Buffered IO会受益颇多。

3) 覆盖写底层IO行为

我们再来看一下Buffered IO模式的覆盖写对应的底层行为。如果文件本身已经有一定尺寸，其长度并不是0，当覆盖写入其中某些区域时，此时因为内核已经可以将文件的对应区域映射到Page Cache中了，那么就会产生写惩罚，读入Page，修改，写回。但是Buffered IO模式下的写惩罚还是与WT模式下有些不一样的地方。

如图19-23所示，使用测试程序对目标文件做从0开始的覆盖写，每

次写1KB内容，同步+连续+Buffered IO模式。还是先来看左侧窗口，可以看到与WT模式相同，产生了写惩罚，但是与图19-6对比，结果是有差异的，即本例中只有惩罚读，并没有惩罚写。惩罚读的倍数与WT模式下相同，惩罚写的倍数为1，即没有写惩罚。可以看到左侧带星号的记录，这些写操作都被缓存在内核中，当程序发起一个FLUSH_BUFFER操作后，内核将这些数据合并为多个Dirty Page，以64KB（16个Dirty Page）为单位，发向CIFS Server。右侧的结果也对应了这个结论，前面是一系列连续的读操作（省略了近千行）而没有任何写操作。而当程序发起FLUSH_BUFFER操作后，Server才接收到以64KB为单位的写操作，然后又接收到了FLUSH Request，这个Request对应了程序最后所发出的CLEANUP操作。

图19-23 以1KB为单位覆盖写

读惩罚、写惩罚、惩罚读、惩罚写

注意： 这里请一定注意这几个词的区别和用法。读惩罚和写惩罚是指在程序发起读操作和写操作的时候，系统底层产生了不必要的额外IO，而这些不必要的IO可能有读，也有写，产生的惩罚IO为额外读，那么就叫惩罚读；如果额外的IO为写，那么就称惩罚写。比如WT模式下的不规则小块写入，会产生写惩罚，写惩罚里既有惩罚读，也有惩罚写。

从结果可以判断，内核的智能有待进一步优化。比如，应当暂存写IO到一定程度然后再判断是否需要读入Page，而不是每接收到一个上层的写IO就先去执行Readpage。如果内核在这方面优化的足够彻底，本实验中是不会产生任何惩罚读的。

对于随机+Buffered IO+覆盖写+小块属性的写IO，由于数据合并为连续的Dirty Page的几率非常小，所以其产生的惩罚写和惩罚读几乎相同，与WT模式无异。

对于Network IO来说，应用程序需要在选择使用缓存和不使用缓存方面谨慎决定。综上所述，一般情况下使用CIFS时最好使用NoBuffer模式以杜绝任何读写惩罚，并且同时保证iosize为4KB的整数倍，这样可以杜绝从Client到Server端整个IO路径中的惩罚现象。仅仅在以下情况考虑使用Buffered IO模式：小块连续读、小块连续写。至于WRITETHROUGH模式，完全是鸡肋，建议不要使用。

另外，使用Network IO的程序还需要适当降低对IO响应时间的要求，还有，在网络异常中断或者服务端异常重启之后需要重新Open()对应的目标文件以便获得最新的File Handle。在其他包括调用接口以及程序设计等方面均与本地IO无异。

提示： Windows系统下的Network IO操作是由操作系统内核的Network IO Redirector（NIR）完成的。Network IO（NIO）相对本地文件系统IO（LIO）的一个最大不同是NIO根本不需要文件—块映射计算，由于NIO底层使用CIFS等协议，这些网络文件系统协议并不像SCSI等块协议一样规定IO Size必须为扇区倍数大小。既然这样，NIR就完全可以透传上层程序的IO请求直接给CIFS Server处理，然后由CIFS Server自身来做文件—块映射运算。所以，在Client端，NIR根本用不到本地文件系统比如NTFS的块映射运算，用到的只是本地文件系统的表示层（见19.1.2节1）功能。NIR管理着NIO层的缓存，这个缓存使用Page为单位进行Page In和Page Out操作（见19.1.4节）。如果用户程序使用Buffered IO模式，则NIR会首先进行Read Cache

Hit或者Cache Write Back操作，如果读未命中，则进行Page In操作读入对应的Page。既然涉及到Page操作，那么NIO对CIFS Server的IO单位就会从原来的不规则程度变为以Page Size为单位对齐规则长度的IO。当然，如果用户程序使用DIO，则可以完全越过系统内核任何的Page Cache，在用户缓存空间与NIR驱动之间建立一条直通路径，这样，程序的不规则长度IO就又被透传到CIFS Server了。

如图19-24所示为NIR在系统IO路径中所处的逻辑位置以及其与内核其他组件之间的关系。

图19-24 NIR 在系统路径中的位置

4) 多进程访问缓存一致性问题解决办法：Opportunistic Lock

本节前文所叙述的全部都是在单Client单进程访问目标文件时所发生的情况。而网络文件系统的最大好处是可以同时让多台Client共同访问同一个文件，当然这不应该算是个好处，本来就是它应该做到的，因为CIFS/NFS等协议本身就是本地文件系统访问接口的一种网络化透传实现方法。既然本地文件可以被多个进程同时访问，那么网络上的文件也没有理由不可以被多个进程或者多个Client访问。涉及多进程同时访问，就一定会涉及到缓存一致性问题。

对于本地文件的共享式访问方法有多种，具体如下。

- 第一种最普通，就是在Open()的时候什么也不做，不使用额外的参数，默认方式下，目标文件是可以共享读写访问的，即可以同时被多个进程打开并读写。显然这样做有很大的安全问题以及数据同步问题，如果使用Buffered IO方式，每个进程都会有自己的

缓存。比如在读数据的时候，内核会将所请求的数据复制到用户程序缓存，如果有多个程序都读取这个文件，那么每个程序自身缓存都会被复制入对应的被请求数据，这些缓存在用户程序缓存中的数据随后可以被程序多次参考而无须再向内核发起读请求；而此时这个程序并不知道，可能有其他的程序早已修改了目标文件的一些数据，而这些被修改的内容并没有实时地被更新到自身的缓存中，自身正在参考的是早已过期的数据。

- 第二种共享访问方法高级一些，程序在Open()目标文件的时候可以指定Share Mode，比如不允许其他进程对目标文件进行读、写、读或写、删等操作，相当于对目标文件加了一个全局锁。这样，加锁的程序就可以肆无忌惮地在程序自身缓存空间对文件进行读或者写缓存而不用担心其他人中途对文件进行修改。
- 第三种更高级，由于全局锁定一个文件会导致其他进程根本无法访问文件的任何部分，有时候某个程序只是想要更改文件中的一小部分地址上的数据，此时程序可以使用内核提供的Lockfile()功能，对文件进行以字节为单位的而不是整个文件的锁定，想要操作哪些字节段，就加锁哪些字节段。多个程序可以共同加锁一个文件不同的字节段，加锁后的字节段可以复制到程序缓存进行读写，然后适当时刻批量向内核发起IO请求以永久保存到底层存储介质，这样效率就有了整体提高。
- 第四种最高级，程序可以使用mmap()功能来将目标文件映射到程序自身缓存但是并不复制其内容，程序针对映射缓存的读写就等于读写了目标文件。多个程序如果都使用这种mmap()功能进行共享映射，那么目标文件在任何时刻只有一份复制，处于系统内核当中，任何用户程序在任何时刻所看到的文件内容都是一样一致的，在mmap()的基础上，程序依然可以使用Lockfile()来对文件进行Byte Range加锁以排开其他进程的访问。

- 第五种，也就是将要介绍的Oplock，见下文详述。

缓存和数据一致性有多个层次，有时序上的一致性（保证每个程序所读入的都是最新的源数据），也有逻辑上的一致性（保证多个进程轮流对某一区域进行更改而不是同时并行写入）。只有Lockfile()才能够保证最高级别的一致性，即时序和逻辑上共同的一致性。也就是说同一个文件的同一个区域，任何时刻只能有1个进程对其做写入操作，多个进程同时对同一区域写入时，必须轮流串行地进行。既然多个进程访问同一个文件区域，那么这多个进程一定都知道自己在做什么，谁先谁后都是进程之间决定好的，如果本来应该后写入的进程却先写入了，那么后来的写入就会覆盖之前的数据，造成逻辑上的不一致。如果不使用Lockfile()，如果进程之间通过另外的机制来实现串行写入，也是可以的，但终究不保险。而对于使用共享式mmap()进行文件访问的多个进程，虽然此时可以保证读时序上的一致性，但如果不使用Lockfile()，一样也无法保证写逻辑上的一致性。

而对于网络文件系统，同样需要解决多客户端访问同一个目标时的缓存一致性问题。网络文件系统的处理方法基本上就是本地文件系统处理方法的透传。网络上每个Client端内核的NIO层缓存就相当于使用本地文件系统IO时单台机器上每个应用程序自身的缓存，而对于CIFS Server端来说，每个Client端就仿佛是一个用户程序，这些程序所发送的文件操作请求会由Server端的NIR模块翻译为针对Server端本地文件系统的请求。CIFS协议其实本质上是将大部分本地文件系统的操作API翻译封装到网络数据包里了。

CIFS协议处理多客户端共享同一目标文件的方法包含了上文描述的四种。第一种就是没有任何限制的自由访问模式，这种模式下根本不能保证缓存一致性。第二种也就是ShareMode模式全局锁，当Client端程序

在Open()时指定这个参数内容之后，CIFS协议会将这个信息放到网络数据包中传递给Server端，Server端NIR驱动解析之后做对应的处理动作，安全透传程序的请求给Server端。如图19-25所示为一个CIFS协议的Open()操作数据包内表示ShareMode的字段，可以看到这个程序允许其他程序对目标进行读和写但是不允许删除操作。

图19-25 Share Mode字段

第三种为Byte Range Lock方法，
程序对某个远程文件发起Lockfile()操

图19-26 字节Lock字段

作之后，CIFS协议完全透传这个请求给Server端。如图19-26所示为一个程序发起的Lockfile()请求被封装在数据包之后的表示，可以看到这一个数据包中包含了三段字节锁，每个长度都是1B，这是用MS Office程序打开一个Word文档时抓取的，被锁的1B看来是文档中的关键字节。

第四种方式，也就是任何时刻，目标文件只有一个副本，CIFS方式下要做到这一点，唯有让所有访问这个目标文件的程序都使用DIO模式打开，这样所有程序所做的更改都会即刻被体现到CIFS Server端内核缓存中。这样的话就可以保证Server端任何时刻只保存目标文件的一份复制，在Server端内核之外没有任何缓存的数据。

网络IO与本地IO最大的一个不同就是本地IO的调用处理过程都发生在内存中，只有底层数据块读写发生在比较慢速的存储介质处，而网络IO过程的调用处理过程本身需要通过慢速网络发送到Server端处理，实体数据的读写也要经过同一个网络，相对于本地IO来讲效率是比较低的。既然这样，如果在客户端使用缓存则会大大降低一些不必要的网络传输但是却无法保证缓存一致性。当然，如果程序都使用Lockfile()，那么不会有任何问题，即使缓存在客户端也不会出现任何不一致情况

(Lockfile())之后，缓存依然不允许Write Back，只允许Read Ahead Hit)，但是不是任何程序都去考虑字节锁的，有些程序不需要逻辑上一致，而只需要在时序上一致即可。使用DIO模式更加直截了当地避免了时序一致性问题，但同时可能增加网络流量。不想加锁又要同时实现客户端本地缓存，还要保证时序上的一致性，似乎挺复杂。

面对这种情况，内核提供了另外一种独立的缓存一致性管理方法，称为Opportunistic Lock (Oplock)，强行翻译过来就是机会锁，当然也可以叫它撞大运锁。Oplock为Windows内核的功能，而并不局限于网络IO，多个进程本地IO也可以使用Oplock来申请程序自身缓存一致性保证。

试想，如果某个时刻只有一个CIFS客户端打开文件进行操作而其他客户端都没有打开这个文件，那么此时这个文件的内容可以被缓存在这个客户端上，可以Read Hit也可以Write Back，不存在缓存一致性问题。

而如果在这个客户端尚未Close()这个文件之前，另有一个客户端也向Server发起针对这个文件的Open()请求，那么Server端必须通知第一个打开这个文件的客户端将其缓存中的Dirty Page全部Flush到Server端，因为有另外的人要看，需要最新的数据。同时第一个客户端也被告知它将不再允许Write Back（每次写都需要直接将数据发向Server端，类似DIO模式），而Read Hit还是可以的，因为第二个客户端此时并没有向文件中写入任何内容，第一个客户端缓存中的数据依然是最新的。随后，客户端2打算向文件中写入内容，那么在写入内容之前，Server会通知第一个客户端其缓存全部作废，因为客户端2已经发起了写操作，已经不能保证客户端1的缓存中都是最新的数据了，客户端1接到这个通知后的IO行为会转变DIO模式，完全Bypass掉内核的缓存了。

同样，客户端2从一开始就以DIO模式行动了，因为此时已经有两个客户端同时读写同一个文件，任何一方也不能保有缓存，双方均以DIO模式运行（一种情况除外，就是如果某客户端向Server申请了Lockfile()字节锁，则对应的字节段可以被读缓存，但依然不能被写缓存）。以上这种思想就是Oplock的设计思想，即Oplock充分保证在只有一台客户端打开文件的时候能够给其充分的缓存自由，一旦有其他客户端打开文件，但是没有做写入操作时，所有客户端都有保有Read Hit缓存也就是读缓存的自由，一旦某个客户端尝试写入文件了，那么所有人的读缓存自由也被剥夺，全部以DIO模式运行。所以称其为撞大运锁，如碰巧你是第一个来的，那么你就能获得缓存，如果再有人来，对不起，写缓存剥夺，如果其他人也要写，对不起，所有人都别缓存了。

注意：一切Write Back缓存在没有电池保护的内存的情况下均会导致Down机之后的数据丢失，Network IO的缓存也不例外。Oplock只能保证数据时序一致性，不能保证数据不丢失。

Oplock在Windows 7之前的操作系统有4种，Windows 7开始增加了4种。这里只介绍前面4种中的3种：Batch Oplock()、Exclusive/Level1 Oplock（RW Lock）、Level2 Oplock（ReadOnly Lock），第4种也就是Filter Oplock不做介绍（因为笔者也没读懂它的具体规范）。

注意：Oplock的申请是由Network Redirector自行向CIFS Server端进行的，一般在用户程序调用Open()时，NIR会自动在发向CIFS Server端的Open请求包中请求Batch+Exclusive的Oplock，Server在应答包中给出审批结果，客户端的NIR根据结果来判断是否使用缓存、读写还是只读缓存。

然而，用户程序也可以主动发起Oplock申请，不过只能够在打开本地文件时才可以，也就是说Oplock在多进程共同访问同一

个本地文件时也可用。但是要注意一点，申请本地文件Oplock时，用户程序必须使用异步IO方式打开文件，因为锁的申请和批准以及断锁通知过程必须使用异步通知机制。本地Oplock的申请和批准过程都是在本地内存中进行的。网络Oplock则需要将信息通过网络发送到Server端申请。

客户端程序直接申请CIFS Server上文件的Oplock是不被允许的，NIR会自动为程序申请。

（1）Exclusive/Level1 Oplock

申请到这种锁的客户端可以进行Read Ahead/Hit以及Write Back模式的缓存，读和写都可以在本地缓存内进行，在适当的触发条件下，Dirty Page才会被批量Flush到CIFS Server端，节约了不必要的网络流量。Exclusive Oplock又被称为Level1 Oplock。

（2）Level2 Oplock

相对于Level1 Oplock，这种Lock在效果上下降了一级。如果客户端申请到的是这种Lock，则表示目前有多个客户端同时打开了目标文件，但是并没有客户端尝试修改这个文件，所以所有客户端都可以保有读缓存，缓存中的内容与Server上目标文件的内容一致。

（3）Batch Oplock

申请到这种Lock的客户端是最幸运的，因为不仅读和写等操作可以在客户端本地缓存中执行，就连Open()、Close()都可以在本地执行。一些批处理脚本经常会在执行脚本的每一行时都对目标进行打开操作，执行完后就关闭，然后在执行下一行时再次打开关闭，如果批处理脚本有

太多行，每次打开关闭操作就会耗费太多不必要的网络流量。Network Redirect在申请到Batch Oplock之后，用户程序每次针对目标文件的打开和关闭操作不会再被发送到CIFS Server端处理，而是由NIR拦截并擅自做了响应。Batch Oplock在级别上与Level1 Oplock相同，只不过后者不能缓存Open()和Close()。

（4）Oplock申请与批准过程示例

这里只举出Batch Oplock申请和批准交互实例，对于Exclusive Oplock的交互过程相比Batch Oplock除了在客户端Open()、Close()时必须发送到Server端之外，其他没有任何区别。如图19-27所示为几轮Oplock申请和批复的动作流程。

图19-27 Oplock申请和批准过程

① 客户端A向服务端发起Open()请求打开文件“test”，并同时申请Batch Lock。

② 由于目前客户端A是唯一一个请求打开这个文件的客户端，所以服务端把Batch Lock给客户端A。

③ 客户端A发起读请求。

④ 服务端返回读请求的数据，这些数据缓存在客户端A，随后针对同地址的读请求可以不必发向服务端。同样，写请求也不必立即发向服务端。

⑤ 客户端A随后对这个文件进行的Open()、Close()、Seek()等操作全部可以在缓存中执行，由Network IO Redirector向程序返回成功响应，不再发向服务端。

⑥ 客户端B向服务端发起Open()请求打开“test”文件，在Open()请求的Access Mask中并没有指定任何写Flag，只有读Flag，同时也在请求中申请Batch Lock。

⑦ 服务端暂挂对B的响应，首先发送一个Lock Break请求数据包给A，并在这个数据包中通知将锁降级为Level2 Oplock。（如果步骤⑥中的打开操作指定了任何与写相关的Flag，则本步中只会Break Lock而不会同时灌以Level2 Lock。）

⑧ 客户端A接收到Lock Break通知后，如果当前这个文件处于打开状态，并未Close，则客户端A会将Write Back缓存里的Dirty Page写入Server，然后向服务端回应一个Lock Break响应包，或者不回复响应包而直接向服务端发起Close()，根据程序设计决定。Close()之后就不会保有目标文件的任何缓存了。如果A接收到Lock Break通知时，目标文件已经处于Close状态，由于Batch Lock模式的缓存不会将之前的Close()发送给服务端，所以接收到Lock Break通知后需要由Network IO Redirector向服务端再发起一个Close()操作以表示接受。

⑨ 本例选择了步骤⑧里的第二种情况，A发送Close()，服务端返回响应。

⑩ 服务端继续处理步骤⑦中暂挂的对B的响应，将响应发送给B。由于A已经在步骤8)中Close了目标文件，所以当前只有B一个人预打开这个文件，所以服务端在响应中灌以Batch Lock给B。

⑪ 客户端A再次发起Open()请求打开目标文件，并在请求中置了写相关Flag，同时请求Batch Oplock。

⑫ 服务端暂挂A的请求，同时发送一个Lock Break通知给B，通知中并没有将锁降级为Level2 Oplock，因为当前A的Open()请求中有写入期望。

⑬ B接收到断锁通知后执行Dirty Page Flush操作，然后可以返回一个Lock Ack给服务端（Lock Ack基本上是Lock Break数据包的复制），或者干脆发送一个Close()给服务端表示不再操作这个文件。

⑭ 服务端继续处理A的暂挂Open()请求。如果上一步中客户端B以Close()回应，则响应中会灌以Batch Lock给A；如果上一步客户端B以Lock Ack回应，则表明B尚未关闭目标文件，那么针对A的响应就不会灌以任何Oplock。

如图19-28所示为另外一个Oplock交互过程。其中在客户端C发出Open()之前，A和B都已经拥有了Level2 Oplock。但是客户端C的带有写Flag标记的Open()操作将一切美梦打碎，服务端强行收回了A和B的Oplock，C自身也没有得到任何Oplock，最终所有客户端的IO请求都被透传到服务端执行。

图19-28 Oplock申请和批准过程

注意： 针对Level2 Oplock的断锁通知，服务端都不会要求Lock Ack。而针对其他类型的Oplock的断锁通知，服务端都要求一个Lock Ack或者Close()。因为Level2 Oplock所缓存的都是源文件尚未改变的信息，通知发过去之后可以不再理会。但是其他类型的Oplock缓存中可能有尚未Flush的Dirty Page，所以，在客户端Flush Dirty Page（一个或者多个Write()请求）之后，服务端一定还需要一个Lock Ack来感知客户端已经完成了所有Dirty Page的Flush工作，然后才会继续处理其他客户端的

Open()请求。

如图19-29所示为一轮Oplock Break过程的4个数据包。1.1.1.3 (A) 这台客户端已经打开了服务端1.1.1.1上的1.rar这个文件并取得了Batch Oplock，随后，1.1.1.2 (B) 也发起打开请求，我们的数据包Trace过程从这里开始。

图19-29 Oplock申请和批准过程实际数据包交互

在178号数据包中我们可以看到这个Open()请求（CIFS协议中Create()与Open()使用同一类数据包）中明确请求Batch Oplock（包中没有与写相关的Flag，图片未截出）。但是由于A已经打开这个文件尚未关闭，所以服务端需要首先向A发送一个Lock Break请求，反映在179号包中（注意被置1的Flag含义）。

注意： 请注意加亮的字段，“Oplock Level Level 2Oplock currently held by client (1)”，意思也就是原来是Batch Oplock，服务端要求A降级为Level2 Oplock。由于Break的是Batch Oplock，所以服务端需要等待A的回应。A的回应反映在180号包中。可见中途并没有发生A向服务端的Write()，证明A自从打开文件之后并未对其进行任何修改。180号包是一个Lock Ack包（内容基本都是Lock Break包的复制）而不是Close包，证明A此时还不想Close()，所以服务端只能对B也灌以Level2 Oplock。181号包为服务端对B的响应，注意加亮的字段：“Level II Oplock Granted (3)”。

(1) 或者 (3) 不表示数量，而是字段值，根据协议规定，每个值代表一种Oplock Level。

Lock Break通知/应答信息与Lockfile()字节锁信息被封装的数据

包使用的是同一种数据包即SMB_COM_LOCKING_ANDX，在图19-23中所示的是这个数据包中最后的部分字段，这个字段包含了字节锁的偏移量和长度。在图19-25中所截取的则是这个数据包另外的字段，这个字段给出了当前Oplock的操作信息。

5) 关于Oplock Break过程的一些细节

在确定是否Break掉Level2 lock之前，Server需要谨慎判断，尽可能地不Break。比如，Server首先检查目标文件的Share Mode，如果第一个客户端Open()时不允许其他客户端写入而只允许读，那么等于给文件加了全局锁，此时如果有其他进程尝试打开这个文件而Open()时在Access Mask中指定了某项写操作Flag，那么Server就不会允许这个Open()操作（返回一个共享冲突错误），从而也不会去Break第一个客户端的Exclusive Lock或者Level2 Oplock（虽然协议是这样设计的，但是另外的根据实验结果判断，实际至少在WinXP系统并未这么做，而是掠过检查ShareMode直接Break其他客户端的Level1或者Batch锁）。

如图19-30所示，图中显示了Open()数据包中包含的Access Mask的一系列Flag；而如果某个客户端在Open()时只附带了只读Flag，那么Server不但不会Break其他客户端的Level2 Oplock，同时也会给这个客户端灌以Level2 Oplock，但是如果已用拥有Level2 Oplock的任何客户端发起了写请求，那么Server会立即剥夺所有客户端的任何形式的Oplock。

图19-30 Share Mode对Lock Break过程的影响

另外，在赋予某个客户端Level2的时候，Server端不会检查当前其他已经Open()目标文件之后的客户端之前在Open()时是否指定了Write相

关的Flag。一般情况下Server端在Break Level1 Oplock之后都会在Break的同时赋予Level2 Oplock，而且对于新执行Open()操作的客户端也会赋予Level2 Oplock（有其他客户端已经打开目标的时候，否则会直接赋予其Level1 Oplock），但是一旦服务端发现随后有任何客户端（包括当前拥有Level2锁的客户端自身）发起写请求时，Server会将之前赋予的Level2 Oplock也剥夺掉。

如图19-31所示，1.1.1.3（A）客户端某个程序正在对服务端的文件“1”进行持续的写入，此时我们在客户端1.1.1.2（B）上打开这个文件进行读操作，服务端与B之间的数据包Trace从这里开始。

图19-31 Oplock申请和批准过程实际数据包交互（1）

1号数据包表示B明确要求Batch Oplock。2号数据包表示服务端并没有因为A在对目标文件持续地写入而拒绝给B任何Oplock，相反，服务端给了B一个Level2 Oplock，很够意思了。

但是在眨眼间，由于A持续地向服务端发起目标文件的写入请求，在把Level2 Oplock给B之后，服务端接收到的第一个A的写请求，将导致服务器收回刚才给B的Level2 Oplock。所以我们看见11号包，是服务端发给B的Lock Break，其中标明了剥夺B的一切锁。

前文中我们说过，当一切锁被剥夺之后，客户端读写缓存被完全屏蔽，IO行为与DIO模式无异。我们看一下图19-31最下部，两个方框中，上面方框表示Lock尚未被剥夺之前的系统IO Size显然为Page Size，而之后，却完全透传上层的IO Size，这里是512B的连续读IO。前文中也说过，读缓存只对小块连续IO效果最大，这里由于读缓存的失败，造成了网络流量升高，而客户端的实际吞吐量却并未增加，网络流量所升高的

部分都由底层网络传输协议的消耗贡献。

我们再来看一下当客户端B掺和进来之后，客户端A上的行为变化。如图19-32所示，原本客户端A上的程序是在以IO Size=2048B针对目标文件做连续并且持续写入动作的。当B发起Open()之后，我们看11062号包，服务端当然要剥夺A上原本的Batch Oplock，并且同时灌以Level2 Oplock。11068号包表示A接受了断锁请求，现在A只有读缓存，写缓存被失效。

图19-32 Oplock申请和批准过程实际数据包交互（2）

但是好景不长，眨眼间，在11073号包中，服务端无情地剥夺了A的Level2 Oplock，为何呢？正如前文所述，拥有Level2锁的客户端对写是很敏感的，一旦有任何一个客户端尝试对目标进行写入操作，那么整个范围内所有客户端上的Oplock将发生坍塌，全部被剥夺。11073号包与图19-31中的11号包是由服务端在坍塌过程中同时向所有客户端发出的。

如图19-33所示，11068号包之后，Batch Oplock被剥夺，只剩读缓存，A上程序的写IO请求被完全透传到服务端；而之前却不是。很显然，之前有严重的写惩罚存在（见上部方框内），而之后，写惩罚消失了。

图19-33 Oplock申请和批准过程实际数据包交互（3）

6) NIR的字节锁缓释行为以及对Oplock Break的影响

在下面的疯狂实验中，我们将调查NIR的另一个怪诞行为，即笔者乱起的一个名字“字节锁缓释”。NIR的这种行为，其流程如下（有客户

端A和B以及服务端)。

A首先打开目标文件并取得Batch Olock。

随后A对目标文件的某些部分使用Lockfile()加入字节锁，按理说NIR应该将这些加锁请求发送给服务端的，但是由于此时是A独占这个文件，所以NIR认为没有必要将这些锁请求发到服务端，而是擅自暂存了这些锁请求。如果随后A解锁了其中某些或者全部的锁，那么NIR也会将这些解锁操作与当前暂存的加锁操作做抵消。

随后，B也申请打开这个文件，服务端首先向A发送锁降级通知，降级为Level2只读锁，那么此时因为Lockfile()操作被内核视为一种对文件的独占操作，现在有其他客户端要来读这个文件，那么A上的NIR必须将这些暂存的锁发送到服务端执行。服务端接受了这些锁请求之后，A才会发送Oplock Break Ack，之后服务端才可以响应B的打开请求。这就是字节锁缓释行为。

这个实验使用分别处于两台客户端上的MS Office Word程序先后打开同一个位于服务端的文件，分析整个过程中的关键CIFS数据包以调查NIR以及Oplock行为。

在客户端1.1.1.3 (A) 首先打开目标文件，并使用Filemon过滤驱动工具监测底层行为，发现Word程序每次都会对文件的固定处进行一系列加锁和解锁过程，如图19-34所示（图中去除了一些非关键记录），可以算出经过加解锁过程之后，程序最终保留的锁只有3个，即偏移量2147483539、2147483559和2147483599三处的长度为1B的锁。但是在本时间段的CIFS数据包中却并没有出现任何发向服务端的加解锁请求，证明NIR驱动擅自暂存了这些请求。我们在接下来的分析中可以发现NIR

对最终的三个有效锁的缓释行为。

随后，客户端1.1.1.2（B）也申请打开这个文件并申请Batch Oplock，同时B的Word程序在打开时指定了Access Mask里面的与写相关的Flag，以及在Share Mode里面也指定了排他性访问，只允许其他进程读取。如图19-35所示，在B的这个Open()请求中，充满了B上的Word程序想要以独占形式打开这个文件的欲望。因为B此时并不知道A已经打开了这个文件，所以B的目的注定不能得逞，因为此时A已经以排他性模式打开了这个文件，所以服务端一定会返回共享模式冲突的通知。

图19-34 程序加字节锁的过程

图19-35 B预使用独占方式访问目标文件

如图19-36所示，216号包。很遗憾，服务端并没有按照Oplock的本来设计思想而首先去判断是否B的这个打开请求与其他已经打开这个文件的客户端存在共享冲突，而略过了这一步直接打算通知A断开它的Batch Oplock，降级为Level2 Oplock。

图19-36 A将隐瞒的字节锁发向服务端

而217号包为A对服务端的回应，我们看到这个Lock数据包中表示的东西并不是一个Lock Break Ack，其中所包含的是A向服务端申请的三个字节锁，也就是说服务端向A通知断开Batch锁，A却先不应答反而向服务端申请字节锁。这里与上文遗留的那三个锁对应，A上的程序早在打开文件之初就做了一系列加解锁过程而最终保留了这三个锁，由于NIR的擅自暂存而一直没被发送到服务端。

现在，A知道服务端要剥夺它的Batch锁了，所以NIR不能再隐瞒了，只能乖乖地将暂存的三个锁发向服务端执行，对应217号包。218号

包为服务端对字节锁请求的回应，我们不去看它。我们来看219号包，这个包就是一个Lock Break Ack包了，其接受了Batch锁降级为Level2锁的通知。

在服务端剥夺A的锁之后，可以对B的打开请求做应答了。如图19-37所示，果然以STATUS_SHARING_VIOLATION告终。但是我们的B依然穷追不舍，在230号包中再次发送了Open()请求，这次，B吃一堑长一智，不在Access Mask中设置任何写相关的Flag，但是Share Mode中依然设置了排他性的只读模式，即不允许其他进程写，B依然抱有幻想它可以独占打开这个文件，所以它的幻想依然还要破灭，230号包显示服务端再次拒绝。

图19-37 B先后三次尝试打开目标文件

但是B上的程序是很有毅力的，它感知到只在Access Mask里妥协是不行的，只有全部妥协才可能有机会，所以B又一次发起Open()。这次它在Share Mode里允许其他进程读和写，见235号包相比230号包的变化部分。236号包中服务端返回了成功响应，这一次，B终于如愿以偿获得了服务端给的File Handle。

但是遗憾的是，服务端并没有给B任何Oplock，见236号包。这似乎有点不可思议，因为前文中曾经提到过，服务端给出Level2锁时并不判断之前其他客户端的打开请求中是否带有写相关的Flag，但是这次为何服务端没有给B呢？原因就在于A上被释放的字节锁。任何字节锁均会导致除加锁客户端之外其他所有客户端的Oplock锁坍塌，有锁的交出来，没锁的不再给。接下来我们就会看到，客户端A的Level2锁随后也因为此被剥夺了。

打开了文件之后，B上的Word程序也需要对文件进行字节区段加解锁过程（至于为何加锁锁哪些区域我们在此不管它，程序设计决定），如图19-38所示。

图19-38 B对文件加字节锁引起A上的Level2锁坍塌

图19-38中上半部分是在B上检测的Word程序实际所做的操作，下半部分为这些操作被发送到服务端时的监测数据，可以看到二者一一对应，除了一个包，247号包。服务端发给了客户端A一个Lock相关的包，可以看到图中最底部对应这个包的内容，是一个Level2锁断开通知。所以，B对文件的字节锁操作，会导致A上Level2锁的坍塌，至此所有客户端都没有Oplock锁了。

当B对文件进行了字节锁操作之后，B上的Word程序会提示一个窗口让用户选择，如图19-39所示。

图19-39 B上提示打开冲突

这里可能有些需要解释的地方，按理说程序应该在收到STATUS_SHARING_VIOLATION之后立即提示这个窗口，但是在此笔者也不知为何程序会在底层尝试最终打开文件并且做了字节锁操作之后才提示。在此猜测可能因为程序需要保证用户选择了只读打开后，程序必须可以打开，所以程序会预先打开测试，如果能打开才提示。这当然只是猜测。

注意： 如果一个客户端打开了文件，获取了Oplock，如果在CIFS Server上某个程序通过本地文件系统也打开这个文件，那么Server会按照同样的策略来Break Lock。另外，如果同一台客户端上的同一个程序先后两次打开一个文件，那么第一次获

得的Batch Oplock将被降级为Level2 Oplock，第二次打开时获得Level2 Oplock。也就是说服务端并不管打开目标文件的进程是不是同一个，或者进程是服务端本地进程还是客户端进程，统一对待。

已经被剥夺的锁不可能自动再获得，比如当所有客户端都关闭目标文件，只剩下一个客户端时，服务端不会再将锁重新灌给这个客户端，协议中没有这样的设计。要想申请锁，必须重新Open()一遍才有可能获得。

客户端可以在使用Oplock的基础上使用Lockfile()字节锁。前者是靠运气而且是全局的，后者是注定的、局部的。

总的来讲，对于CIFS方式的Network IO模式下，系统内核缓存的效率很低，多数情况下都是因为惩罚值太高而帮了倒忙，所以我们还是不用为好。Oplock只在特定情况下能够发挥一点作用，程序设计人员需要注意，要时刻知道底层都在做什么。本书的目的就是让读者对整个存储系统架构有一个深刻理解，以方便今后的管理维护和开发工作。

7) NFS下的缓存和IO机制

同为NAS网络文件访问协议，NFS不管在数据包结构上还是在交互逻辑上相比CIFS要简化许多，但是简化的结果就是并不如CIFS强大，CIFS之所以复杂是因为CIFS协议中几乎可以透传本地文件系统的所有参数和属性，而NFS携带的信息很有限。简化同样也带来了高效，执行类似的操作，NFS交互的数据包在单个包的大小上和整体发包数量都相对CIFS有很大的降低。

与CIFS不同的是，NFS提供诸多更改的参数来控制操作系统内核底层IO行为。Windows系统可以安装SFU软件来实现NFS的客户端或者服

务端功能，同样也提供了少量可更改的参数。

在了解NFS底层行为之前，我们先看一看NFS方面的4个比较典型的参数设置。这4个参数中，前三个都是在mount的时候可以指定的，包括sync、rsize、wsize，第四个参数O_DIRECT是需要程序对文件打开调用时指定的。下面我们分别来调查这4个参数对系统底层IO行为的影响，但是首先要来看一下默认mount参数下NFS底层行为。测试基于RHEL5系统，内核版本2.6.18。

(1) 默认mount参数下的IO行为

NFS默认的mount参数为async、rsize=wsize=65536。async参数表示系统内核不会透传程序的IO请求给NFS Server，对于写IO会延缓执行，积累一定的时间以便合并上层的IO请求以提高效率，不管是读还是写请求，async都会具有一定效果，尤其是连续地址的IO。

如图19-40所示，上半部分为使用dd if=/mnt/3 of=/dev/null bs=1500 count=100来读取NFS文件，使用底层抓包工具抓取的底层IO结果，下半部分为使用dd if=/mnt/3 of=/dev/null bs=150 count=100时的结果。Linux下的dd是一个使用同步调用+Buffered IO模式的程序，但是在图中上半部分却显示出明显的底层异步IO行为，即IO批量发送，数据批量返回。这种IO行为是由于文件系统的预读造成的（这里并不是读IO被合并的结果，见下文）。可以看到，由于dd进行的是连续读操作，所以导致了文件系统最大力度的预读，每次发向存储设备的IO字节数为65536，也是Linux下rsize的最大值。可以算出系统共读入了 $409600 + 65536 = 475136\text{B}$ 的内容，但是dd所读取的一共是 $1500 \times 100 = 150000\text{B}$ ，所以，系统预读造成了 $475136 - 150000 = 325136\text{B}$ 的浪费。再来看下半部分，系统共读入了81920B，dd请求了 $150 \times 100 = 15000\text{B}$ ，浪费掉

81920-15000=66920B。

图19-40 dd测试结果

还记得CIFS么？相比NFS来讲，在连续读的情况下，CIFS低效得多，CIFS根本没有做主动预读的意思，只有被动预读。

我们再来看图19-41，使用测试软件对NFS文件做随机读操作，每次请求200B的数据，可以看到，读惩罚产生了。随机读情况下，内核底层读入了1个Page的内容。这里需要提一下，对于本地文件系统，Linux同样使用Page Cache映射方式。在Windows下，CIFS在随机读情况下也产生类似的读惩罚，但是二者有本质不同：Windows下的CIFS的读惩罚是因为Page Fault过程，底层必须以Page为单位；而Linux下NFS的读惩罚只是因为文件系统预读的单位为Page。二者虽然表象相同，但是其出发点却不同。

图19-41 产生惩罚现象

我们再来看看写行为。写有两种方式：追加写和覆盖写。如图19-42所示，使用`dd if=/dev/zero of=/mnt/3 bs=1 count=100`来对/mnt/3这个文件进行追加写操作，可以看到17号包对应的SETATTR请求，dd程序将已经存在的文件的size设置为0，然后从头开始追加写入这个文件，每次写1B，写100次。但是内核在底层可并没有向NFS服务端发起100次写，实际上只发送了一个写请求，对应19号包。offset=0，length=100，也就是从头开始写100B，证明内核将dd发送的写请求数据全部缓存并且合并了起来发送给NFS服务端。

图19-42 写合并的发生

图中最底部的部分是在使用`dd if=/dev/zero of=/mnt/3 bs=1500 count=1000`时抓取的底层IO结果，可以看到dd为同步调用，到了底层，内核将dd的写IO数据合并，并且以异步的方式高效发向NFS服务端。

注意： 仔细的人可以注意到，590号包的Length并不是65536，这是因为合并之后的数据本身就并不对齐，所以最后系统将余量完全透传到NFS服务器，而并不会发生读入一个对齐的单位，修改之，然后再写入惩罚现象。

再来看一下覆盖写。覆盖写又可以分为两种：连续覆盖写，随机覆盖写。这里要理解，追加写一定是连续写，而随机写一定是覆盖写。

如图19-43所示，使用测试工具对一个NFS文件进行以1B为单位的覆盖写入操作，上半部分为连续写情况下的底层IO监测结果，而下半部分则为随机写的监测结果。

图19-43 覆盖写测试结果

可以看到，连续覆盖写情况下，内核同样是对写操作进行了合并操作。而随机写情况下，内核根本无法合并写IO，所以内核的做法是完全透传用户程序发起的IO，而且，没有任何写惩罚。

还记得Windows下的CIFS么？在Windows下使用CIFS，患有严重的惩罚综合症，不但读有，写更严重。Linux下使用NFS，对于写操作，不管offset是否为Page或者512B对齐，都没有任何写惩罚的存在，而对

于读操作，也只有在随机读的情况下出现了读惩罚，其他任何情况下都没有惩罚。

（2）指定sync参数后的IO行为

默认mount使用的是async模式，内核可以对IO进行合并以提高效率。这里有一点需要理解，即内核处理的async或者sync与上层程序调用时的sync与async紧密相关。如果上层使用sync调用，则其产生的读IO一定在内核处也是同步执行的，因为只有在前一个读请求数据成功地被返回给程序，程序才会发起下一个读请求；对于异步读调用，内核可以在短时间内接收到多个读请求，此时内核可以将这些读请求合并处理，这就是异步过程了。对于同步写调用，程序只有在前一个写操作完成之后才会发起下一个写，而如果程序使用了Buffered IO模式，那么一个写操作会迅速瞬时完成。因为内核只要将待写数据复制到系统内核缓存即可通知程序完成，继而接着发起下一个写操作，那么内核在短时间内即可积累若干待写数据，这些数据并不是没接收到一次就必须向底层存储介质写一次的，此时内核可以合并这些写，这也是一个异步过程。而对于上层的异步写调用，内核处理过程就更是异步了。所以，内核的异步过程，只对Buffered IO模式下的同步写、异步写、异步读有意义。

我们这就来将内核的这种行改一下，不让它缓存IO，要求它立即执行每一个IO。在mount的时候我们使用mount -o sync来挂载一个NFS Export。然后分别使用dd程序对目标文件进行1500B为单位的读操作和以1B为单位的写操作。如图19-44所示，上半部分为读操作的监测结果，下半部分为写操作的监测结果。

图19-44 sync参数的影响

我们可以看到，对于读操作，底层是不是似乎将程序发起的读IO合并了呢？根本不是，上文曾经分析过，除非程序自己使用异步读IO调用，否则内核底层不可能在单位时间内接收到多个由同一个进程发起的读请求，所以也就无从合并。出现这种现象的根本原因是文件系统预读行为导致的。`sync`参数表示IO需要被立即执行而不得延迟，但是对于读立即执行和写立即执行，内核却有不同的处理方式。读立即执行，不一定表示不可以进行Prefetch和Cache Hit操作，但是对于写立即执行，却绝对不可以将待写的数据缓存起来延迟处理。正因如此，图中下半部分说明内核对写IO不缓存、不延迟，再接收到之后，立即透传给了NFS服务端处理。

8) 指定`rsize/wsize`参数后的IO行为

NFS客户端在mount服务端的exports时，服务端会向客户端通告服务端自身所能接受的最大IO Length。客户端根据这个Length来决定，如果用户程序发起的IO Length大于这个值，那么NFS客户端有责任将程序的IO分割为小于等于这个值；如果小于这个值，可以直接透传，也可以合并成这个值来发送。

使用`mount -o rsize=1024`，我们将`rsize`从默认的64KB改为1KB，看看底层会发生什么事情。首先使用`dd`产生连续读IO来触发内核的预读操作，如图19-45所示，内核预读依然发生，但是向底层存储系统发起的IO Length降为1KB，效率大大降低。

图19-45 `rsize`参数的影响

如图19-46所示，使用测试程序产生小块随机读操作，此时内核会产生读惩罚，针对每个读操作都会读入一个Page。但是由于`rsize`的改

变，本来一个IO就可以读入的Page，现在必须被分割为4个IO才可以读入。注意方框中的offset。

图19-46 读惩罚产生

使用`mount -o wsize=1024`将wsize改为1KB，使用dd程序进行连续写入操作，内核虽然合并了这些写IO，但是无奈，合并的IO最终还会被分割成1KB的IO发送到NFS服务端，如图19-47所示。

图19-47 wsize参数的影响

可以看到，改变rsize/wsize，对NFS的IO逻辑没有任何影响，受到影响的只是底层传输的数据包数量和大小。任何情况下均不要降低rsize或者wsize，百害而无一利。这个参数的调节只是为了使用NFS服务端的要求而已。

注意：在RHEL5以及其他一些Linux系统上，如果使用`mount -o rsize=32KB`这种命令格式来更改rsize或者wsize的话，那么系统虽然在执行命令之后没有任何错误信息报出，但是系统内核在底层却并没有按照给出的这个值来分割IO请求，而是统统使用了4KB为单位，如图19-48所示。

图19-48 底层使用4KB为分割单位

而如果使用`mount -o rsize=32768`这种命令格式，此时就没有任何问题了，如图19-49所示。虽然32KB=32768B，但是内核似乎并不认同前者，使用时一定要倍加注意。分割单位在不知情的情况下变小，会增加存储系统负担。

图19-49 底层使用命令给出的值为分割单位

9) 程序使用O_DIRECT参数后的IO行为

Linux下的NFS比Windows下的CIFS优异，表现在，前者有明显预读力度，只在特定条件下有读惩罚，写惩罚一点没有。正因NFS的缓存如此高效，所以在Linux 2.6内核中，在mount时并没有提供Direct IO的选项（内核编译时被禁止），但是单个程序在Open()的时候依然可以指定O_DIRECT参数来对单个文件使用DIO模式。与Windows下CIFS实现方式相同，如果选择使用了DIO模式，那么NFS层就会完全透传程序层的IO请求。如图19-50所示，以DIO方式读取文件，每次1B，底层完全透传了上层的IO请求。

图19-50 DIO模式测试结果

10) 多进程访问下的缓存一致性解决办法

在缓存一致性的保证方面，NFS相比CIFS来讲要差一些。CIFS使用Oplock机制来充分保证文件的时序一致性；而对于NFS，除了使用字节锁或者干脆使用DIO模式之外，没有其他方法能够在使用缓存的情况下严格保证时序一致性，NFS只提供“尽力而为”的一致性保证，而且这种保证全部由客户端自行实现，NFS服务端在这个过程中不作为。下面我们就来看一下NFS提供的尽力而为的一致性保证机制。

比如，有两个客户端共同访问同一个NFS服务端上的文件，而且这两个客户端都使用本地NFS缓存，那么客户端A首先打开了文件并且做了预读，而且A本地缓存内还有被缓存的写数据。此时客户端B也打开了这个文件，并且做了预读，如果被读入的数据部分恰好是A被缓存的尚未写入的部分，那么此时就发生了时序不一致。而这种情况在CIFS下是不会发生的，因为B打开时，服务端会强制让A来将自己的写缓存

Flush，然后才允许B打开，此时B读入的就是最新的数据。

再回到NFS来，B上某进程打开了这个文件之后，内核会将文件的属性缓存在本地，包括访问时间、创建时间、修改时间、文件长度等信息，任何需要读取文件属性的操作，都会Cache Hit，直到这个Attribute Cache (ac) 达到失效时间为止，如果ac达到了失效时间，那么内核NFS层会向服务端发起一个GETATTR请求来重新取回文件最新的属性信息并缓存在本地，ac失效计时器被置0，重新开始计时，往复执行这个过程。在ac缓存未超时之前，客户端不会向服务端发起GETATTR请求，除非收到了某个进程的Open()请求。其他诸如stat命令等读取文件属性的操作，不会触发GETATTR。

在任何时刻，任何针对NFS文件的Open()操作，内核均会强制触发一个GETATTR请求被发送至服务端以便取回最新的属性数据。这样做是合理的，因为对于Open()操作来说，内核必须提供给这个进程最新的文件数据，所以必须查看最新属性以与ac缓存中的副本对比。如果新取回的属性信息中mtime相对于本地缓存的信息没有变化，则内核会擅自替代NFS服务端来响应程序的Open()，并且随后程序发起的读操作也都首先去碰缓存命中，不命中的话再将请求发给服务端，这一点类似CIFS下的Batch Oplock；但是如果新取回的文件属性中对应的mtime比缓存副本晚，那么就证明有其他客户端的进程修改了这个文件，也就意味着本地的缓存不能体现当前最新的文件数据，全部作废。所以此时，内核NFS会将这个Open()请求透传到服务端，随后发生的读写数据的过程依旧先Cache Hit（由于之前缓存作废，所以第一次读请求一定是不命中的），未命中则从服务端读取。随着缓存不断被填充以最新读入的数据，命中率会越来越高，而且直到下次出现同样的过程之前这些缓存的文件属性副本和数据副本不会作废。

ac的超时值是可控的，在mount的时候可以指定“actimeo=秒数”这个参数来设定。当然，这个值设定的越低，客户端发送GETATTR到服务端的频率也就越高，就能更快地感知到目标文件的变化。如果将其设置为0，意味着不缓存文件属性信息，那么客户端上的进程每次查询文件属性的操作（比如stat命令）均会触发一个GETATTR被发送到服务端。

Mount NFS时还有一个选项叫做“noac”，即No Attribute Cache缩写，这个选项与actimeo=0的区别是，noac等价于“actimeo=0”+“sync”。

对于读操作，使用noac或者actimeo=0选项的情况下，如果程序没有使用DIO，那么内核依然有预读行为。但是，为了确保程序读入的数据是最新的，每次程序发起read()，如果内核判断这个read()命中了当前的缓存，则会同时触发getattr发向服务端。如果取回的属性mtime未变，则从缓存中将数据返回给程序；如果mtime有变，则向服务端发起read()取回最新数据。如果程序发起的read()未命中，则无须触发GETATTR，直接向服务端发起read()取回数据。这种做法的好处是极大地利用了预读缓存，每个read()用一个GETATTR为代价来探寻缓存的文件属性是否为最新，一旦走运为最新，那么即可不需要向服务端发起read()。

如果使用DIO模式，每个程序发起的read()就均会被映射为内核向服务端发起的read()，享受不到预读缓存的效果。使用上面这种方法，虽然每个read()均会导致发送一次GETATTR，但是GETATTR消耗的资源相比read()来讲是很少的。但是这样做依然对性能有很大的影响，如果整个环境中只有一台客户端，那么使用noac是严重浪费网络 and 系统资源的，一个进程可能在单位时间内发起大量的read()操作，就算是这些read()均命中了缓存，那么依然不能避免内核在底层会发向与read()调用

等同数量的GETATTR请求数据包给NFS服务端。网络上可以看见成千上万的GETATTR，这是多此一举的行为，缓存的提速作用都被外部网络的速度给制约了，根本没有体现出来，所避免的仅仅是被假设的可能需要从服务端读数据的过程。

CIFS的Oplock模式一旦某个客户端锁被剥夺，那么除非进程再次Open()，否则不可能再使用本地缓存。但是NFS下可以随时使用缓存，只要GETATTR取回的信息中mtime未变。就算mtime变了，本地缓存作废，之后还会积累新的缓存数据可供Read Cache Hit，直到下一次作废为止。

总结：Linux下的NFS缓存一致性解决办法，从严格到不严格依次为程序使用DIO模式、使用noac选项来mount、降低actimeo阈值到尽量低的值、默认mount参数。按照这个顺序，客户端获得的性能是递减的，DIO模式根本无法使用缓存，noac模式只能靠发送GETATTR来碰运气使用预读缓存，actimeo超时周期内则可以肆无忌惮地使用本地缓存而不需要向服务端发起GETATTR来探询。默认mount参数的actimeo=60，也就是1分钟，所以缓存发挥的作用更大。GETATTR是多客户端访问环境下NFS实现缓存一致性的法宝。

11) Linux上使用CIFS与Windows上使用NFS

我们知道，Linux上现在已经可以直接以CIFS协议来mount一个CIFS服务端的共享目录，同时Windows上也可以通过安装Microsoft提供的Windows Services for UNIX (SFU) 开发包来实现对NFS服务端的mount。现在我们来调查一下这两者的一些行为与原配情况下有什么异同。Linux使用RHEL5，自带了对CIFS的支持；Windows使用WinXP SP3，安装SFU 3.5开发包。

我们首先调查Linux上使用CIFS的情况。使用默认参数`mount -t cifs`命令来mount服务端的一个共享目录。如图19-51所示，图中第一项结果对应用dd进行同步+Buffered IO模式从目录某文件中读出1byte的内容的过程；第二项结果对应用dd进行同步+Buffered IO模式每次1B读入10000次的过程；第三项结果对应使用测试程序进行Buffered IO+同步调用模式以1B为单位持续随机读取文件的过程；第四项结果对应使用测试程序进行Buffered IO+同步调用+WriteThrough模式以1B为单位持续随即覆盖写入文件的过程。

图19-51 Linux下使用CIFS

从上图的各项测试结果来看，Linux下的CIFS访问方式同样存在读写惩罚，究其原因还是由于Page Cache导致。看来Linux下除了使用NFS可以避免惩罚之外，本地文件系统和CIFS文件系统均无法避免读写惩罚。Linux下还是使用原配的NFS最好。

我们再来看一看Windows上使用NFS访问的情况。如图19-52所示，第一项结果对应使用测试工具进行Buffered IO+同步调用模式下以1B为单位对文件进行随机读的过程；第二项结果对应使用测试工具进行Buffered IO+同步调用+WriteThrough模式下以1B为单位对文件进行随机写的过程；第三项结果对应用测试工具进行Buffered IO+同步调用模式下以1B为单位对文件进行连续地址写的过程。

图19-52 Windows下使用NFS

是否产生读写惩罚与操作系统的具体实现有关系，不能将这种惩罚归于某个协议的缺陷，协议自身是没有这种惩罚缺陷的。每个协议作为

一个模块嵌入操作系统的驱动链，产生惩罚的原因在于OS与模块之间的接口设计。

Windows与Linux对待本地文件都使用Page Cache映射方式，Linux对块设备也使用Pcache映射。两个操作系统均有读写惩罚现象。Windows对待NTFS与CIFS文件系统是一视同仁，而Linux下则进行有区别的对待，充分考虑了NFS网络IO的效率等因素。但是遗憾的是Linux与CIFS之间的接口尚待完善。而Windows与NFS模块之间则为彻底的DIO模式，完全透传，这也不失为一种快刀斩乱麻的实现方式。

19.1.2 文件系统层

文件系统是系统IO路径中首当其冲的一个比较大的模块。IO离开应用层之后，便经由OS相关操作被下到了文件系统层进行处理。文件系统一个最大的任务就是负责在逻辑文件与底层卷或者磁盘之间做映射，并且维护和优化这些映射信息。文件系统还需要负责向上层提供文件IO访问API接口，比如打开、读、写、属性修改、裁剪、扩充、锁等文件操作。另外，还需要维护缓存，包括预读、Write Back、Write Through、Flush等操作；还需要维护数据一致性，比如Log、FSCK等机制；还需要维护文件权限、Quota等。

可以把一个文件系统逻辑地分为上部、中部和下部。访问接口属于上部；缓存管理、文件管理等属于中部；文件映射、一致性保护、底层存储卷适配等属于下部。

1. 文件系统上部

文件系统对用户的表示层处于最上部，比如Linux下的表示

法“/root/a.txt”、“/dev/sda1”、“/mnt/nfs”或者Windows下的表示法“D:\a.txt”、“Z在192.168.1.1上的Share\1.txt”。文件系统表示层给用户提供了一种简洁直观的文件目录，用户无须关心路径对应的具体实体处于底层的哪个位置，处于网络另一端，还是磁盘上某个磁道扇区。这种FS最顶层的抽象称为Virtual File System（VFS）。

文件系统访问接口层也位于上部。由于接口层直接接受上层IO，而IO又有14种（见表19-1），文件系统会受到这14种IO的摧残。面对这种摧残，文件系统还是有点对策的。在Windows系统下，NTFS提供了两种适配IO类型的API：FILE_FLAG_SEQUENTIAL_SCAN和FILE_FLAG_RANDOM_ACCESS。在应用程序打开一个文件时可以在调用相关函数的时候给出对应的Flag。前者表示应用准备对这个文件进行连续读写（LBA地址连续）访问，如果使用这个参数而且使用系统内核缓存，那么文件系统会加大默认的预读力度和频率，这个参数在不使用系统内核缓存时是无效的；后者则表示应用准备对这个文件进行随机读写访问，那么文件系统就会降低默认的预读力度和频率以避免不必要的浪费。

我们在此做一个“疯狂实验”来折磨一下文件系统，笔者用Asynchronous Explore进行使用系统内核缓存的随机IO，但同时又指定了连续IO参数，通告FS程序要进行连续IO但实际却给出随机IO。这显然是矛盾的。我们利用这个矛盾来调查文件系统是否也跟着我们矛盾。如图19-53所示，在抓取的iSCSI包结果中我们可以看到，在一开始瞬间的21号包表示FS向底层存储发起了一个长度为128B的读IO，这显然受到了那个参数的影响，随后的几个长度为128B的IO表明FS接连预读了几次。随后，再也没有发生超长度IO，表明FS从此不再进行预读了（后续还有数百个IO，图中没有截出）。这说明FS是很有智能的，它一开始

参考给出的参数，但是随后它发现上层给出的IO并不是连续的，受骗了，所以愤然不再预读。

图19-53 验证FS预读行为

2. 文件系统中部

缓存位于文件系统中部。预读和Write Back是文件系统的最基本功能，可以参考上文中的示例来理解文件系统预读机制。缓存预读对不同IO类型的优化效果也是不同的，参考表19-2。

表19-2 不同IO类型的缓存效果

OS可以提供一些可供用户自行设置的参数来控制文件系统预读力度，比如AIX系统便提供了j2_maxpagereadahead参数来控制JFS2文件系统的预读力度，这个参数的值表示FS做预读操作时向底层发起的IO Size。比如，将其设置为4则表示每个预读IO Size为4个缓存Page，也就是 $4 \times 4\text{KB} = 16\text{KB}$ 。下面我们用数据来说明FS预读的效果。首先看一下将j2_maxpagereadahead改为4之后，cp一个文件所耗费的时间以及底层向存储系统发起的IO情况。其中Read Ops表示每秒读操作数，Read KB表示每秒读带宽吞吐量。

```
aix-cn22:/#ioo -o j2_maxPageReadAhead=4
Setting j2_maxPageReadAhead to 4
aix-cn22:/#mount /dev/fslv02 /mnt3
aix-cn22:/#time cp /mnt3/c /mnt2
real    0m17.07s
user    0m0.28s
```

sys		0m11.00s						
Read	Write	Other	QFull	Read	Write	Average	Queue	Lun
Ops	Ops	Ops		kB	kB	Latency	Length	
3994	0	0	0	63888	0	0.00	0.08	/vol
3116	0	0	0	49872	0	0.00	0.07	/vol

我们看到cp耗费了17秒，底层预读发起的IOPS为3000~4000之间。再来看一看将j2_maxpagereadahead改为64之后的结果。可以看到cp耗费了11秒，大大加快了，而且底层预读耗费的IOPS仅为400左右，差不多是前者的十分之一，同时带宽也达到了测试用的1Gb/s的iSCSI以太网链路的极限。

```

aix-cn22:/#ioo -o j2_maxPageReadAhead=64

Setting j2_maxPageReadAhead to 64

aix-cn22:/#mount /dev/fslv02 /mnt3

aix-cn22:/#time cp /mnt3/c /mnt2

real    0m11.67s
user    0m0.22s
sys     0m9.34s

```

Read	Write	Other	QFull	Read	Write	Average	Queue	Lun
Ops	Ops	Ops		kB	kB	Latency	Length	
383	0	0	0	97856	0	1.04	0.03	/vol
437	0	0	0	95936	0	0.90	0.03	/vol

我们再来看一看图19-54，这个实验中，在Windows系统中使用测试工具对某个文件使用以200B为单位的连续读操作，NTFS文件系统此时

的预读力度达到了最大，左侧显示了系统底层发向存储设备的IO，每个IO的长度几乎都是128，即64KB。

图19-54 FS Aggressive Prefetch

右侧显示了测试程序所在的操作系统底层的行爲，可以看到7号记录是一个由于读惩罚而产生的惩罚读（带星号记录，凡是由于Page Fault导致的Page In操作，都带星号），但是后续再也没有出现惩罚读，因为需要读入的数据早已被预读，不再产生Page Fault。

对于应用程序的写IO操作，文件系统使用Write Back模式提高写IO的响应速度。这种模式下，应用程序的写IO数据会在被复制到系统内核缓存中之后而被通告为完成，而此时FS可能尚未将数据写入磁盘，所以，如果此时系统发生Down机，那么这块数据将丢失，而应用程序可能并不知道数据已经丢失而造成错乱的逻辑，可能造成严重的后果。这种担忧在关键业务应用中是绝对要杜绝的。NTFS文件系统提供了另一个参数：FILE_FLAG_WRITE_THROUGH，只要程序在打开某个文件时给出这个参数，那么程序的写IO数据依然会先被复制到系统内核缓存，文件系统随即立即将这份数据写入磁盘，然后向程序返回完成的信号。这个参数与FILE_FLAG_NO_BUFFERING不同，后者表示读和写IO均不使用缓存，而前者读IO依然使用缓存。所以，数据库类程序启动时在打开Log文件时都会给出这个参数。Write Back又可以称为Delay Write或者Lazy Write，即数据并不是立即就被写入底层存储的。Write Back模式除了会丢失数据这个致命缺点之外，剩下的全是优点，比如，大大降低IO响应时间，还有大大增加写命中率。

提示：何谓“写命中”？读命中率大家都理解，即待读取的数据

已经存在于缓存中而无须从磁盘读入缓存。而对于写IO，每个写IO数据都将首先被放入缓存，然后才会被写入磁盘，岂不是每个写IO都会命中缓存么？哪里来的“命中率”呢？试想这样一种情况，T1时刻有针对地址LBA1的B1数据待写入，B1进入缓存后，由于Write Back机制，B1尚未被写入硬盘之前，T2时刻应用又发起了针对同地址的写IO操作B2，那么此时FS是否有必要让B2来占用新的缓存空间呢？随后当FS需要将数据写入硬盘时，先写入B1，然后又写入了B2，这是没问题的。但是如果先写入了B2，后写入了B1呢？这时候就是大问题了。实际上，FS在接收到B2数据后，会将B2覆盖B1，因为从时间历史角度看，B1已经不复存在，在随后的写盘动作时，只需要一次而不是两次写入动作，节约了后端IO资源。这便是写命中，即待写入的数据对应的地址在缓存中恰好存在之前的尚未被写盘的IO数据。

文件系统还使用另外一种IO优化机制，叫做IO Combination。假设T1时刻有某个IO目标地址段为LBA0~1023，被FS收到后暂存于IO Queue中；T2时刻，FS尚未处理前一个IO，此时又有一个与第一个IO同类型（读/写）的IO被收到，目标地址段为LBA1024~2047。FS将这个IO追加到IO Queue末尾。T3时刻，FS准备处理Queue中的IO，FS会扫描Queue中一定数量的IO地址，此时FS发现这两个IO的目标地址是相邻的，并且都是读或者写类型，则FS会将这两个IO合并为一个目标地址为LBA0~2047的IO，并且向底层存储系统发起这个IO，待数据返回之后，FS再将这个大IO数据按照地址段拆分成两个IO结果并且分别返回给请求者。这样做的目的是节约后端IO资源，增加IOPS和带宽吞吐量。每发起一个IO，SCSI协议层以及底层传输协议层都会有相应的开销，如果能够将本来需要多次发送的IO合并为一次IO，那么对应的协议

开销就会避免。

3. 文件系统下部

文件系统下部包括文件—块映射、Flush机制、日志记录、FSCK以及底层卷接口等相关操作。

位于FS下部的一个重要的机制是文件系统的Flush机制。在WB模式下，FS会暂存写IO实体数据与文件的Metadata。FS当然不会永久地暂存下去而不写入磁盘。文件系统会在适当的条件下将暂存的写IO数据写入磁盘，这个过程叫做Flush。有多种条件可以触发Flush：距上次Flush时间、某个缓存Page在缓存中待的最长时间、应用强制触发的Flush、FS自身为了实现某功能（比如快照）而自行Flush以及其他各种原因。

同样，位于文件系统的下部还有文件映射处理模块。在NTFS文件系统中，一种被称为Sparse File的处理机制可以大大节约文件实际占用磁盘的空间，而且还可以加快读写速度。某些文件本身非常大，但是文件中包含的数据大部分都是二进制0，如果考虑压缩这些0从而降低实际物理占用空间，NTFS自身的压缩机制是可以做到的，但是压缩带来的一个坏处就是在读写被压缩数据的时候需要耗费太多的系统资源，影响速度。取而代之，FS提供另一种机制：Sparse File。如果某个应用程序认为自己生成的文件符合Sparse的特性，那么程序可以将这个文件设置为Sparse模式，对已有文件和新建文件都可以这样操作。已有的文件设置为Sparse模式之后，程序需要显式地将其中全0的部分使用相关函数通告FS，FS会将这些部分存放在一个特殊的列表中备查。当某个程序向Sparse文件中写入大片连续0的时候，FS会感知到并且将对应的地址追加到列表中而无须真正写入磁盘，只有当待写入的数据为非0时才会被真正写入磁盘。当读取Sparse File的时候，FS会首先查询列表，如果发

现匹配的地址则直接返回0给应用程序而无须从磁盘中读。

在很多情况下，某些应用程序需要创建一个文件但是暂时不向其中写入数据，一般这种情况下所创建的文件大小为0B，占用磁盘空间仅为元数据所占用的空间，一般初始时为一个文件系统块比如4KB大小，此时文件并没有实际内容，所以文件本体被保存于元数据中存放。随着程序对文件的写入，文件会逐渐增大。但是，在其他一些情况下，程序希望预先让这个文件占用一定的空间以防止随后一旦出现磁盘空间不够时的尴尬，为了实现这一点，程序可以用Writefile()逐渐向文件中写入0一直到期望的大小为止，或者使用SetEndOfFile()（NTFS）和Setfilevaliddata()之类的函数来扩充文件的长度。

调用不同的方法，会得到不同的结果，拿NTFS举例，每个文件有三种长度属性，也就是标称长度、分配长度和逻辑占用长度。标称长度没什么可说的，分配长度就是文件在磁盘上实际分配的占空大小，而逻辑占用长度则是指文件自创建之后被实际写入到哪个长度（注意，不是被实际写入了的总长度，而是实际写入地址的高水位线）。

如果调用Setendoffile()，比如用这个函数把文件大小从0字节设置为1GB字节，那么OS会为这个文件在磁盘上分配实际的1GB空间，也就是在文件分配表记录中分配实际的空间指针，并且同时会更新文件所在的卷的bitmap文件，但是却不会向每个被占用的扇区写0，所以这个动作会瞬间就完成，此时这个文件的标称长度和实际分配长度都是1GB，但是逻辑占用长度是0，因为还没有任何数据被写入，此时如果发起针对这个文件任意地址的读操作，OS会在内存里生成0x00返回给应用程序，不需要读盘。但是随后如果发生任何针对这个文件任意偏移地址的写操作，比如向其位于500MB处的地址写入1MB的数据，那么OS会在后台同时将这个文件的0字节开始一直到第500MB之前的所有空间写

0（写到该地址所占用的磁盘扇区里），但是501MB到文件尾部的那499MB空间不会写0，同时设定该文件的逻辑占用长度为501MB，如果发生任意针对501MB之前偏移地址的读写操作，OS均会读写磁盘，如果发生501MB到1GB之间地址的读操作，OS不会读盘，会在内存中生成0x00，写操作，则重复刚才的动作，在高水位线之前的未被填0的所有地址上填0。所以，如果某个应用预先创建了一个大文件占位之后，后续如果发生针对这个文件的随机写操作，那么OS势必会在后台发生很多填0操作，会非常影响性能。

提示： 诸如迅雷等下载工具，一般都会预先占用文件空间，但是由于迅雷针对文件的下载并不是顺序的，会有多个线程从文件的多个部分开始并行写入，此时如果使用Setendoffile()来占位，性能就会很差。但是可以使用另一个函数也就是Setfilevaliddata()，这个函数与Setendoffile()的区别就是OS任何情况下都不会在后台向文件中填0，这样做的好处是明显的，但是也会带来安全性问题，为文件分配了对应大小的扇区但是不向这些扇区中写入0，那么这些扇区之前的内容就会变为这个文件当前的内容。所以，除非程序明确知道自己在做什么，否则稀里糊涂地使用这个函数可能造成驴唇不对马嘴的后果。通常使用这种方法的典型例子就是下载工具。

大家可以做一个试验，使用迅雷下载一个大文件，将其保存在一个曾经被塞满过的分区上。迅雷在获取到这个文件的大小之后会立即创建相应大小的文件，而创建文件的过程是瞬时完成的，并没有长时间磁盘IO操作。此时我们立即退出迅雷。查看这个文件的属性，发现其大小和磁盘占用空间相同。然后用十六进制编辑器打开这个文件，可以发现其中并非全0，而是被塞满了凌乱的数据。这充分证明迅雷使用的就是

SetFileValidData()函数来扩展文件大小的。随着文件不断地被下载，新的数据将会覆盖这些扇区原有的内容。

笔者曾经遇到过更有趣的现象。某个分区中曾经存有多部电影，随后删除掉一些。某次笔者使用迅雷下载一部新电影存放到这个分区，但是下到70%左右没有速度了，于是强行播放下载的文件，奇迹发生了：在电影播放到某个时间点处，突然出现了另一部电影的畫面，这部电影恰好就是笔者之前删除的那些的其中一部。这个巧遇绝对属实，但是由于懒惰，没有做重现试验。

1) 文件分布映射

由于底层存储卷也是由物理磁盘经过层层虚拟化操作而生成的，最终影响IO性能的根本因素和最大因素还是物理磁盘的机械寻道。RAID的做法可以利用起所有RAID组成员磁盘的寻道时隙来提高总体性能（附录1的问题8），但是这种做法具有很高的盲操作性（参考本书4.2.6一节）。想要让一个RAID Group发挥出最大的并发IO性能，就首先需要保证RAID控制器接收到的针对这个RAID组的一个IO或者一批IO可以一次性发送给组内的所有磁盘进行操作。将一个IO发散给多个磁盘操作贡献为系统带宽吞吐量增加，而将多个IO同时发送给组内磁盘操作（每个磁盘操作一个IO）则贡献为系统IOPS增加。

提示： 为了实现增加系统带宽吞吐量的目的，RAID 3将Stripe Size调到最低，比如一个FS的IO单位大小，这样就可以盲性地保证每个FS层面的IO都一定会被分散在组内所有磁盘上。如果FS发起的是连续IO类型，那么这种情况整个系统的带宽吞吐量将会非常高，这也是RAID 3唯一表现优秀的情况。而如果FS层面发起的是随机小块的IO，这时系统性能将会非常差。因为

RAID 3注定一个组一次只能处理一个IO操作（因为每个IO都要占用所有物理磁盘），大量随机的IO不但得不到并发，而且加上磁盘寻道的影响，造成所有物理磁盘在处理下一个IO时总要全体寻道，加之现代缩水版RAID 3并不保证组内磁盘转速同步，造成磁盘旋转延迟都不同，先达到目标扇区的磁盘会处于等待状态，这又浪费了系统性能。

RAID 3是一个古老的RAID类型。那时的应用程序和业务要求并没有现代这么苛刻。而现代的业务系统对存储的要求越来越苛刻，既要求高IOPS即IO并发度，又要求差不多的带宽吞吐量。为了增加系统IOPS，顺便解决RAID 3不能并发IO的问题，RAID 5出现了。RAID 5相对RAID 3的最大改进其实是增加了条带宽度而不是引入分布式校验，这里很多人的理解都有误区，总以为分布式校验才是RAID 5性能提高的最大因素。为了说明这一点，我们假设只在RAID 3基础上将单Parity盘变为分布式Parity，而Stripe宽度不变，仍为一个FS层面的IO大小（一个FS Block，一般为4KB）。试问，这种情况下，管它Parity是不是分布式，对系统性能有提升么？每次整个RAID组依然只可以处理一个IO，同样都是整条读或者写，Parity放到条带的哪个位置有关系么？根本没有，性能没有一点提升，反而还会由于分布式Parity运算复杂度升高而导致的计算延迟增加。所以，RAID 5相对RAID 3的根本进化在于提高了Stripe Size，可以设置为很大，比如1MB或者更大。在这个基础上，对于小块随机IO的并发度就会大大增加，因为条带增加使得每个磁盘对应的Segment（条带深度）也增加，使得一个IO仅落在一个磁盘上的几率也增加，多个IO同时分别落在一个磁盘的几率也增加，也就是IO可以并发执行了。这一步改进对应RAID 4。然而，RAID 4的读IO可以并发了，但是写IO却依然

不可以并发，因为每次写IO过程中都需要修改Parity盘上的校验值，所以Parity盘就会被这个IO占用，此时其他的写IO只能等待，因为所有写IO都要更改Parity。为了解决这个问题，RAID 5出现了，分布式校验就这样诞生了，其诞生在逻辑顺序上是处于增加条带宽度之后的。改为分布式校验后，写IO也可以并发了，只不过并发几率远低于读IO。

RAID 5的并发几率属于守株待兔型而不是主动型，合理地布好网，猎物是否成群结队地进来就靠运气了。所以这种设计是盲性的。但是RAID 3的做法并不是盲性的而是釜底抽薪型，不管任何情况下，每个IO必定由组内所有盘同时操作，但是这样做也注定了RAID 3的局限性。RAID 5的写IO并发几率远低于读，三块盘的RAID 5根本不能并发写IO，小块写IO性能非常低下。就算增加磁盘，盲并发几率提高也很缓慢，再加上碰运气，实际获得的并发几率低之又低。此时，需要一种机制来增加写IO并发几率。FS下层已经是黔驴技穷了，唯一能折腾折腾的地方只有在FS层面了。如果猎物能够自投罗网，那就是再好不过的事情了。

作者一直认为，文件系统是系统IO路径中的一个很重要的角色，而文件在底层存储卷或者磁盘上的分布算法是重中之重，成败在此一举。传统的文件系统只是将底层的卷当做一个连续扇区空间，并不感知这个连续空间的物理承载设备的类型或者数量等，所以也就不知道自己的不同类型的IO行为会给性能带来多大的影响。

将IO比作昆虫，底层的RAID比作不同类型的蜘蛛网。我们都见过那种藏匿于灌木丛中的密度很大几乎是一块白布样的蜘蛛网，也见过稀稀疏疏孔状的八卦样蜘蛛网，走路时也偶尔会碰到一根蜘蛛丝绕在你脸

上的情况。当然，昆虫永远也不会自己投向蜘蛛网。而文件系统要想获得最优的IO性能，就必须在感知底层RAID类型的情况下按照对应的策略自投罗网。比如，如果蜘蛛网是白布状，那么任何一个小昆虫都逃不过；如果是稀疏孔状，那么过小的昆虫可能漏网；只有一根蜘蛛丝的情况往往用于探测某个信号，比如一个庞大的东西走过。同样，现行的做法就是底层存储设备布网等待上层的IO到来。如果我们在FS层对底层的RAID类型和磁盘类型等各种因素做出分析和判断，然后制定对应的策略，让文件能够按照预期的效果有针对性地在RAID组内进行分布，这样就会和谐。比如，在格式化文件系统时，或者在程序调用时，给出显式参数：尽量保持每个文件只存放在一个物理硬盘中，那么当创建一个文件的时候，文件系统便会根据底层RAID的Stripe边界来计算将哪些扇区地址段分配给这个文件，从而让其物理地只分布到一个磁盘中。这种文件分布方式在需要并行访问大量文件时是非常有意义的，因为底层RAID的盲并发度很低，如果在FS层面手动地将每个文件只分布在一个磁盘上，那么N个磁盘组成的RAID组理论上就可以并发N个针对文件的读操作，写操作并发度仍相对很低但是至少可以保证为理论最大值。

总之，文件系统必须与底层完美配合才能够获取最大的性能。这方面WAFL的做法为文件系统与底层配合的典范，不过WAFL也存在着自身不可避免的问题。关于WAFL机制的简要论述见本书第4章。将来的文件系统设计可能会考虑更多的底层适配因素，让我们拭目以待。

2) 卷IO

卷IO策略处于文件系统的最底层，负责将数据从卷中读出或者向卷中写入。这个层次需要用最少的IO做最多的事情。数据库类程序的Log机制是耗费IO最多的情况之一，每次Checkpoint都会将日志以Write Through模式写盘，由于Bypass了FS的缓存，FS来不及做过多优化，所

以造成对底层的IO过频，影响性能。另外，保证卷IO Queue随时充满、异步操作等，都是榨取底层存储系统性能的有效方式。

19.1.3 卷管理层

卷管理层在某种程度上来讲是为了弥补底层存储系统的一些不足之处的，比如LUN空间的动态管理等。卷管理层最大的任务是做Block级的映射，对于IO的处理，卷层只是做了一个将映射翻译之后的IO向下转发的动作以及反向过程。另外，应用程序可以直接对某个卷进行IO操作而不经文件系统。

注意：这里的不经文件系统并不是说Bypass系统内核缓存的Direct IO，而是完全不需要FS处理任何块映射关系。这时就需要由应用程序自行管理底层存储空间，而且此时不能对这个卷进行FS格式化或者其他未经应用程序允许的更改操作，一旦发生将导致数据被破坏。

卷管理层将底层磁盘空间虚拟化为灵活管理的一块块的卷，然后又将卷同时抽象为两种操作系统设备：块设备和字符设备。比如在AIX系统下，/dev/lv、/dev/fslv、/dev/hdisk等字样表示块设备，而/dev/rlv、/dev/rfslv、/dev/rhdisk等带有r字样的设备一般就是字符设备。同一个物理设备会同时被抽象为字符和块两种逻辑设备。用户程序可以直接对块设备和字符设备进行IO操作。这两个设备也是用于上层程序直接对卷进行访问的唯一接口，有各自的驱动——块设备驱动和字符设备驱动，在IO路径的层层调用过程中，IO Manager访问卷的时候其实就是访问对应的设备驱动（这一点在系统IO模块架构图中并没有体现），向它们发起SystemCall的。

1. 块设备

在UNIX类操作系统下，块设备表现为一个文件，而且应用程序可以向块设备发起任何长度的IO，就像对文件进行IO时一样，比如512B、1500B、5000B等，IO长度可以为任何字节，而不需为磁盘扇区的整数倍。然而，块设备也是由底层物理设备抽象而来的，而底层物理设备所能接受的IO长度必须为扇区的整数倍。所以块设备具有一个比较小的缓存来专门处理这个映射转换关系。

块设备一般使用Memory Mapping的方式被映射到内存地址空间，这段空间以Page（一般为4KB）为单位，所以访问块设备就需要牵扯到OS缺页处理（Page Fault）方式来读写数据。比如应用程序向某个块设备卷发起一个长度为1500B的IO读，卷管理层接收到这个IO之后将计算这个1500B的IO所占用的扇区总数以及所落入的Page地址，并且进入缺页处理流程从底层物理设备将这个Page对应的扇区读入，这里的IO请求为1500B，所以OS会从底层物理设备读取对应的1个Page大小的数据进入缓存，然后从缓存中再将对应的1500B返回给应用程序。

应用程序对块设备发起读IO，块设备就得同时向底层物理设备发起对应的转换后的IO，不管应用程序向块设备发起多少长度的IO，块设备向底层物理设备所发起的IO长度总是恒定的（一般为4KB，即缓存Page大小）。所以块设备向底层物理设备发起的读IO属性永远为小块IO，而且对同一个线程发起的IO不会并发只能顺序，对多个线程共同发起的IO才会并行，也就是说每个线程在底层的IO都为顺序执行（限于读IO）。这一点是块设备非常致命的缺点，比如一个应用程序256KB的读IO操作，会被块设备切开成为64个4KB的读IO操作，这无疑是非常浪费的，会更快地耗尽底层存储的标称IOPS。但是对于写IO来讲，块设备底层

会有一定的merge_request操作，即可以对写IO进行合并、覆盖、重排等操作，这方面内容详见下面的章节。

我们来举例说明一下。下列数据为在AIX系统上使用IO测试工具对一个块设备（/dev/fslv01）进行IO Size为0.5KB、1KB、2KB、4KB、8KB、16KB、513B的单线程顺序连续IO读操作时，在存储系统端统计的结果。其中Read Ops表示每秒读操作数，Read KB表示每秒读带宽吞吐量，Queue Length表示当前LUN（/vol/LUN/LUNaix）的IO Queue中的IO数。我们可以发现，不管哪一组的测试数据，用带宽除以IOPS得出的IO Size恒定为4KB，也就表示主机向存储系统（块设备向底层物理设备）发起的读IO Size恒定为4KB。

其实UNIX类系统下的块设备与文件系统管理下的一个文件无异，唯一区别就是直接对块设备进行IO操作的话，无须执行文件—块映射查询而已。

读操作对于块设备来讲还不至于产生太过恶劣的性能影响，而写IO则会更加严重地摧残存储设备的性能。由于块设备向底层发起的所有IO均以缓存Page大小为单位，现代操作系统的Page一般为4KB大小，如果某应用程序需要写入0.5KB数据，或者4.5KB数据，那么很可怕，块设备不能直接把对应长度的数据直接写入底层设备，而必须先读入这个IO占用的4KB单位Block，然后修改之，然后再将数据写回到底层设备。其浪费可谓是惊人而且无法容忍的！我们来看一个例子。下列数据显示了AIX系统上使用IO测试工具对一个块设备进行4096B、2000B、2048B、5000B写IO时系统底层发向物理磁盘的IO统计情况如图19-55所示，程序发起写IO时，不对齐4KB的IO Size会导致OS首先读入对应的Page数据，修改，然后再写入对应的Page数据。所以可以看到写动作伴随了一定程度的读动作，也就是写惩罚。

图19-55 块设备写IO测试

如图19-56所示，当程序向块设备发起从0地址开始的2000B单位的

图19-56 Page边界对齐

连续写IO时，在接收到第一个IO请求之后，OS内核必须首先从磁盘读出Page1到缓存，然后将前2000B内容在缓存中覆盖为程序所IO的内容，然后再将修改后的Page1写入磁盘：

- OS接收到第二个IO之后，由于第二个IO是请求从1999~3999这段地址，依然落入Page1，所以OS还需要再次读入Page1，但是此时需要考虑以下两种情况。
- Page1当前最新的数据应为第一次IO所修改的数据，如果第一次IO的Page1尚未被写入磁盘而依然在缓存中，那么此时OS必须在缓存中将最新的Page1数据保留为第二次IO使用，同时又不影响第一次IO的Page1的写盘动作。这样，这个读请求就算Cache Hit了，就不需要从物理磁盘读数据。

如果在第二次IO发起之前第一次IO的Page1已经被写入磁盘，那么第二次IO必须从磁盘读入Page1然后执行与第一次IO相同的过程。第三次IO情况就更复杂了，由于第三次IO的地址为4000~5999这段地址，跨越了两个Page，4000~4095地址落入了Page1，4096~5999地址段落入了Page2，那么此时OS内核必须同时读入Page1和Page2，同时修改然后同时写入磁盘。同理，如果IO Size选择为5000B等，还有可能发生同时跨越3个Page的，那写惩罚会更大。

细心的读者可能会发现，即使上层IO不是4KB对齐的情况下，底层

的IO应该也都是4KB对齐的，因为不管读还是写都是以Page为单位，但是为何在图19-55的结果中用KBPS一栏的值除以4却除不尽呢？这个问题很好，底层的IO并不是4KB对齐的原因，是因为在块设备驱动处，OS做了Merge_request，对此下面的章节会有详细描述。

系统IO路径中的读惩罚和写惩罚： 在系统IO路径中，有多处可以发生读惩罚和写惩罚，其中包括文件系统层、卷管理层、块设备层和底层RAID管理层等位置。所谓“惩罚”就是说要完成某件事，必须付出一些额外的牺牲和浪费。到底什么情况下会出现惩罚呢？

读惩罚： 当某个读IO请求的IO Size不可被OS Page Size或者Disk SectorSize任何一个除尽为整数时，这个读请求就会产生读惩罚，但是可被除尽并不代表一定就不产生惩罚。比如某个IO的IO Size为从Offset 0开始读后续的512B，则怎么都好说，退一万步讲，如果这个IO是针对某个文件的，而所请求的这段数据恰好就在磁盘一整个扇区上，那么OS只要将对应磁盘上的这个扇区的内容读出来即可，上下一一对应，没有浪费；而如果这个IO的起始Offset不为0，比如为5，那么就邪门了，FS在分配文件占用空间的时候是以512B对齐的，从0开始往后的512B（包括0）就一定会对应到底层磁盘的1个扇区上，但是如果从5开始往后的512B，就一定不会只落在一个扇区上，肯定是跨两个扇区，那么此时OS就不得不读出这两个扇区来，从其中各自取出对应这个IO的部分，然后合起来回传给程序。并且，OS内核往往都是以Page为单位，也就是读出IO地址段所落入的整个或者多个Page。

具体惩罚的细节与不同操作系统有关，按照系统内核对IO请求的计算处理过程不同，一定条件下是否产生、产生多少程度的

读惩罚随OS不同而不同，随着调用的方式不同而不同。比如同样为1KB的读IO，如果针对FS Block=1KB的文件系统进行越过缓存的DIO，那么就不会产生读惩罚；同样的1KB的IO Size，如果针对某个块设备进行IO，那么就会产生额外3KB的浪费的读（内核会读出底层的1个Page也就是4KB大小的数据来对应这个IO）。

写惩罚：当某个写IO请求的IO Size不可被FS Block、OS Page Size或者Disk Sector Size任何一个除尽为整数时，必定会产生写惩罚。写惩罚程度同样随OS的不同而不同，也随调用方式的不同而不同。写惩罚的表现是既有额外读操作，又有额外写操作，比读惩罚浪费更多的资源。比如某个程序发起一个Offset 0开始的513B的读IO，恰好比一整个扇区多了1B，但是底层又不可能只写1B给磁盘，磁盘接受的必须是扇区整数倍的IO Size，那么此时唯一办法就是写两个扇区，第一个是0~511这512B，然后再写入512~1023这后512B，剩余的那1B就是这后512B的第1个字节，那么对于后512B中减掉这1字节剩余的511B的数据，我们只能先从磁盘将后512字节对应的扇区读出来，将第1个字节内容替换成程序IO所给的内容，然后再将这个扇区写回去。这就是写惩罚，既多了511B的读，又多了511B的写。

读惩罚和写惩罚的例子在本章下面的章节随处可见。

2. 字符设备

传统的字符设备本来是专指一类接受字符流的设备比如物理终端、键盘等，这种设备的特点是可以直接对设备进行最底层的操作而不使用缓存（但是必须有Queue），而且每次IO必须以一个字符为单位（卷所

抽象出来的字符设备以一段连续扇区为一个单位）。所以具有这种特点的实际设备或者抽象设备都被称为字符设备。而将卷抽象为字符设备并不是说将IO从扇区改为字符，而只是抽象出字符设备所具有的特点。

在任何操作系统下，对字符设备进行IO操作必须遵循底层的最小单位对齐规则，比如对于卷字符设备来讲，每个IO长度只能是扇区的整数倍，如果IO长度没有以扇区为单位对齐（比如513、1500），那么将会收到错误通知而失败。虽然UNIX类操作系统下的字符设备也表现为一个文件，但是这个文件却不像块设备一样可以以任意字节进行IO，因为OS没有为字符设备设置任何缓存（但是存在Queue）。

字符设备的一个最大好处是可以发起底层协议允许的（SCSI/ATA，256KB）任意扇区倍数长度的IO而且可以完全透传上层应用程序的IO。所以字符设备可以对底层对应的物理设备发起任何属性的IO，比如大块连续并发IO，这样就非常好地适配了上层应用程序对IO的要求，可以获得很高的性能。字符设备是一个设备最底层的抽象，其本质等于物理设备本身。下面的测试数据说明了上述结论。

从上面的结果可以看到，底层发出的IO长度与应用层发起的IO长度一一对应。

3. 裸设备与文件系统之争

字符设备又被称为裸设备。应用程序可以选择使用文件系统提供的各项功能进行对文件的IO操作，当然也可以选择直接对裸设备进行IO操作，只不过直接对裸设备操作需要应用程序自行维护数据—扇区映射以及预读缓存、写缓存、读写优化等。比如数据库类程序自身都具有这些功能，所以没有必要再使用文件系统来读写数据。而由于块设备的诸多

不便和恶劣性能影响，不推荐直接使用。那基于文件系统的IO和基于裸设备的IO方式到底孰优孰劣呢？我们就此讨论一下。

文件系统拥有诸多优点是毋庸置疑的，但是对于某一类程序，FS提供的这些“方便”的功能似乎就显得很有局限性了，比如缓存的管理等，由于文件系统是一个公用平台，同时为多个应用程序提供服务，所以它不可能只为一个应用程序而竭尽全力服务；况且最重要的是，FS不会感知应用程序实际想要什么，而且FS自身的缓存在系统异常Down机后还容易造成数据不一致情况的发生。其次，使用缓存的IO方式下，对于读请求，系统IO路径中的各个模块需要将数据层层向上层模块的缓存中复制，最后才会被OS复制到用户程序缓存；对于写请求，虽然缓存IO方式下，写数据被OS接受后即宣告完成，但这也是造成Down机后数据丢失的主要原因之一。所以，对于大数据吞吐量IO请求，避免内存中多余的数据复制步骤是有必要的（此外还有另外一个原因在下面章节介绍）。但是对于一般的程序，是完全推荐使用文件系统进行IO操作的。

另外一个最重要的原因，在使用内核缓存以及文件系统缓存的情况下，容易发生读写惩罚，这是非常严重的浪费。

对于这类对IO性能要求非常高而且对缓存要求非常高的程序，它们宁愿自己直接操作底层物理设备，也不愿意将IO交给FS来处理。这类程序的典型代表就是数据库类程序。虽然这些程序也可以使用文件系统来进行IO操作，但是这个选择只会给程序带来一个方面的好处，那就是文件管理会方便，比如可以看到数据文件实实在在地被放在某个目录下，可以直接将数据文件复制出来做备份，做文件系统快照保护等。而选择文件系统所带来的坏处也是不少的，比如最大的劣势就是重复缓存预读，FS预读了数据，数据库程序依然自己维护一个预读缓存，这两个缓存里面势必有很多数据是重复的，增加了许多空间和计算资源开销，而

且这些数据不见得都会产生Cache Hit效果。所以这类程序宁愿使用裸设备自行管理数据存储和数据IO，所带来的唯一缺点就是数据管理很不方便，除了程序自身，其他程序只看到了一块光秃秃的裸设备在那儿，里面放的什么东西，怎么放的，只有程序自己知道。

4. DirectIO与裸设备之争

有没有一种方法能够结合FS和裸设备带来的优点呢？有的。为了既享受文件系统管理文件的便利同时而又不使用FS层面的缓存，将缓存和IO优化操作全部交给应用程序自行处理，FS只负责做文件—扇区映射操作以及其他文件管理层面的操作，节约内存耗费以及提高处理速度。操作系统内核提供了一类接口，也就是前文中出现的FILE_FLAG_NO_BUFFERING参数。当然，这个参数只是Windows内核提供的，其他操作系统也都有类似的参数。这种Bypass系统内核缓存的IO模式统称为DIO，即Direct IO模式。在UNIX类系统下，在Mount某个FS的时候可以指定“-direct”参数来表示任何针对这个FS的IO操作都将不使用内核路径中任何一处缓存。当然，也可以在应用程序层控制，比如打开文件时给出O_SYNC或者O_DIRECT、FILE_FLAG_NO_BUFFERING之类的参数，那么不管目标FS在mount时给出了何种参数，这个程序的IO都将不使用文件系统缓存。

注意： 由于内核的DIO模式只是Bypass了缓存，其他任何接口均未变，所以DIO模式下，内核文件系统允许应用程序对文件做出非DIO模式下相同的任何操作，包括读写任意字节长度的文件数据（Windows除外，见前文），而这种貌似“透明且便利”的接口，恰恰依然还会带来读写惩罚。

注意： 在Windows系统下，如果应用程序选择不使用系统缓

存，则应用程序自身的缓存必须为底层存储介质扇区的整数倍大小（完全按照底层设备要求的IO长度单位，内核在接收到用户程序IO时并不做检查，直接交给底层驱动，如果底层驱动发现这个IO长度并不符合它的要求，那么会一层层向上报错直到用户程序。如果底层驱动为NIR驱动（见上文），那么由于诸如CIFS之类协议规定IO长度甚至可以为0B，所以底层使用CIFS的时候，使用DIO模式的用户程序可以发起不规则长度的IO），因为越过了系统内核缓存，OS将直接将应用程序自身缓存作为底层存储设备的DMA空间。而底层IO设备都要求目标内存空间必须为扇区整数倍，即扇区对齐，所以应用程序在运行时被推荐使用OS提供的特殊函数功能来为自己分配内存最为保险。另外，应用程序发送的IO请求中地址段也必须为下层卷扇区或者块的整数倍（AIX系统下可以发送任意长度IO）。

在Windows系统下，程序如果选择使用DIO模式，那么操作系统会生成一份Memory Descriptor List（MDL），将对应指针传递给底层设备驱动程序并且在IRP（见下文）中通知驱动程序本次IO使用DIO模式，同时锁定应用程序自身缓存中对应本次IO的Page。驱动程序使用MDL直接对应用程序内存进行访问。这也是为何在Windows系统下选择DIO模式的程序自身缓存以及IO Size必须为扇区整数倍的原因。本章后面会介绍Windows下DIO的详细步骤。

Windows下的DIO模式（FILE_FLAG_NO_BUFFERING）是彻底的透传上层请求的（不管FS Block是多少，上层IO请求包含多少个扇区长度，OS会向底层物理设备发起对应的扇区长度的IO），而AIX下则不是。但是Windows下的DIO模式也要求程序发起的IO必须为底层存储介质最小单位的整数倍（CIFS方

式除外，见前文），而AIX下则可以为任意长度。也正因为如此，Windows才可以透传应用层的IO请求，AIX由于允许应用层IO不对齐，所以其底层也不可能透传应用层的IO。

在AIX系统下，虽然OS允许程序在DIO模式下发起任意字节长度的IO，但是由于底层物理设备只能接受扇区对齐的IO长度，像块设备IO一样，这之间存在一个转换关系，内核自动将不对齐的IO转换为底层对齐的IO，完成这个动作需要一小部分的缓存。对于读IO操作，最差的情况为应用程序如果每次IO只读取1B的内容，那么这个IO到了底层便会变为对物理设备4KB的读IO操作，当取回数据之后，OS只返回给应用程序所读取的那1B的内容，然后将读入的4KB内容从缓存中删除，即多耗费了4096倍的IO操作和4096倍的带宽。如果使用内核文件系统缓存，那么这种浪费是不存在的，FS可以直接从预读缓存中将对应的字节直接返回给应用程序。DIO模式与块设备IO模式在多数情况下效果类似。

下面的例子显示了AIX系统下对一个DIO模式Mount的FS进行以0.5KB的IO Size为单位递增的读测试数据。

可以看到，DIO模式对应的底层IO Size并不与应用层IO Size一致，但是DIO模式与块设备IO模式有一些区别。比如块设备IO模式下只有在应用层IO Size=Page Size时，底层与上层的IO才是一一对应的；而DIO模式下，只有在应用层IO Size=FS Block Size时，底层IO才是与上层一一对应的。

与块设备一样，对于DIO模式下写操作最差的情况是，应用程序如果每次IO只写入1B的内容，那么底层会首先读入待写入字节所落入的4KB块到缓存，然后更新对应的字节，然后再将更新后的4KB块写入物理设备，多耗费了底层8192倍的带宽，而且这其中有一半的耗费为写操

作。写比读所耗费的资源更多，雪上加霜，这种情况是不可容忍的。

下面的例子显示了AIX系统下对一个DIO模式Mount的FS进行以0.5KB的IO Size为单位递增的写测试数据。与块设备写IO相同，具有一定规模的写惩罚。

所以，利用DIO模式读写文件或者直接读写块设备的应用程序一定要明确自己在做什么，为何使用DIO，DIO会带来什么。但是只要程序发起的IO的IO Size是FS Block的整数倍，那么就不会浪费资源，也就是说既然选择了DIO，就要在应用层面多考虑一层。虽然可以与访问缓存模式FS同样的方法访问DIO模式的FS，但是需要底层付出巨大代价的，我们必须从源头上消除这种代价。块设备也一样。这方面Windows的做法是强制要求程序自身缓存和IO Size为扇区整数倍，而AIX却没有强制要求，但是开发者必须认识到这一点。所以，如果可以的话，对于数据库类程序尽量使用裸设备进行IO，一了百了。

注意： Write Through与FILE_FLAG_NO_BUFFERING有着本质区别：前者模式下，数据依然首先进入操作系统内核缓存，只不过内核保证数据被写入磁盘之后才返回成功而不是像Buffered IO模式下只要数据进入内核缓存便立即返回成功；而后者则表示数据根本就不进入内核缓存，直接由底层驱动从用户程序自身缓存取走数据从而写入底层存储介质。

这二者看似效果类似，实则有很大不同：前者很有可能依然导致内核Page Fault流程导致的写惩罚；而后者则不会有内核层面的写惩罚。所以开发者需要注意了，使用时最好直接选择后者而不是前者。有些IO测试软件甚至没有使用NoBuffer模式而只用了WT模式，殊不知这种情况下根本无法体现出底层存储系统的真实性能，由于读写惩罚的原因导致所得的结果总是过

低，而此时将存储端的IO监测数据与程序得出的数据比较就会发现，底层存储的实际吞吐量以及IOPS都可能远大于程序所报告的。

关于前者依然产生读写惩罚的问题，前文Network IO一节的疯狂实验中有介绍。

注意： 这里请注意一个概念问题。操作系统内有多处缓存，其实面对用户程序的第一处缓存并不是通常理解的“文件系统缓存”，而是一处被称为“SystemBuffer”的缓存。当用户程序选择使用Cache IO或称Buffered IO时，每发起一个IO请求，操作系统便会根据程序IO请求的数据所占用内存的大小在操作系统内核内存空间同样分配一块与其相同大小的内存用来充当SystemBuffer，然后再往下才是文件系统缓存。任何读入或者写入的数据都需要经过SystemBuffer。而DIO模式下，操作系统会Bypass首个缓存，也就是SystemBuffer，同时也通知IO路径中首层驱动使用DIO模式，首层驱动程序（包括文件系统也是一种驱动程序）直接从用户程序内存中将数据取出（对于写请求）或者送入（对于读请求）。下面的章节中会讲述Buffered IO的详细过程。

5. 关于CIO模式

某些文件系统，比如AIX下的JFS2，采用读共享、写独占的方式来处理多个进程访问同一个文件的情况，即如果多个进程访问同一个文件，如果没有任何进程对这个文件进行写操作，那么所有进程都可以同时读取这个文件的任何内容；但是一旦有某个进程在对这个文件进行写操作，那么其他所有进程都将被禁止访问这个文件，不管是读还是写。

这样充分保证了文件数据的一致性，保证所有进程在相同时刻都会看到相同的内容。这种做法虽然保证了数据一致性，但是FS自作主张的这个决定只是对一些没有考虑到多进程并行访问同一文件情况的应用程序有效果，对于那些自身已经对这种情况考虑足够充分的数据库类程序来讲，FS的这个做法不但是多余的，而且还极大影响性能，使得多个进程不能同时并发写同一个文件而只能串行地进行。而数据库类程序恰好就要求多进程多线程同时写一个文件，比如负责写Log文件的进程或者写数据文件的进程，这些进程往往都有多份复制在同时执行以获得最大的并发度和性能。

针对这个需求，AIX系统下的JFS2文件系统提供了另外一种接口，称为CIO模式，即Concurrent IO，这种模式下FS将不再自行锁定文件，而是完全对多个进程放开对文件的同时访问权限，将保证数据一致性的责任完全交给应用程序执行。开启CIO模式之后，AIX会自动强制开启DIO模式。另外，如果同时打开同一个文件的多个进程中有一个或多个并没有在打开文件时指定使用CIO模式，那么OS就不会使用CIO模式来操作这个文件，多个进程之间仍然必须串行写入文件，当没有使用CIO模式的进程退出后，OS就会自动使用CIO模式来操作文件。还有，当某个进程试图对某个被多进程同时打开的文件进行虚写IO时，也就是没有更改文件实际内容而只是更改一些文件属性，或者更改文件长度等操作时，OS会自动将CIO模式失效而恢复原来的独占模式；当操作完成后，会恢复CIO模式。

注意： Windows的做法与JFS2不同，Windows提供一种指定文件访问方式的API参数，即Share Mode，程序在打开文件的时候可以选择使用何种Share Mode，比如其他进程只读、其他进程不可读写、其他进程可读写、其他进程可删除等。所以

Windows本身就已经可以实现类似CIO的模式，只要每个进程指定可读写Share Mode即可。

图19-57 CIO性能对比

在使用CIO模式之后，数据库类程序的IO性能将会与裸设备IO性能接近。如图19-57所示为CIO、DIO和裸设备IO性能之间的对比曲线，可以看到由于单纯DIO模式下不可并发，造成系统性能相对裸设备差别很大，而CIO模式下则差别较小。

对于数据库类程序，使用AIO+裸设备IO模式为最优的IO模式，可以获得最大的性能。如果考虑文件管理便利性方面，则可以退而求其次使用AIO+CIO模式，也可以在获得很好性能的同时又不失去文件系统带来的好处。

注意：多个进程利用CIO打开文件时，请确定这个文件最好是定长的，即进程在写文件的时候不会将数据追加到文件尾部或者插入文件中或者对文件进行truncate等造成文件大小改变的操作。比如数据库的Log文件，数据库的Log写进程一般都会有多个来并行执行追加写入，如果Log文件是动态增大的，那么CIO丝毫不会起到作用，多个进程也没有存在的意义，所以Log文件都是在创建的时候就被指定了一定长度而且是被重复覆盖写入的。

DIO、CIO、AIO、SIO之间的关系：AIO和SIO是指程序在调用OS相关API之后自身的动作如何，阻塞还是不阻塞；而DIO和CIO是指程序在调用OS相关API之后OS内核的动作如何，是使用缓存还是越过缓存，是并发写入还是串行写入。程序可以使用AIO+DIO、AIO+CIO、SIO+DIO、SIO+CIO这4种组合。

19.1.4 层与层之间的调度员：IO Manager

IO Manager或称IO Scheduler。每个操作系统都会有这样一个角色，它专门负责接受上层程序的IO请求，然后将IO请求下发到对应的模块和设备驱动中执行，然后将结果通知给上层程序。当某个程序试图访问某个文件的时候，它其实并没有直接和文件系统模块打交道，而只是在与IO Manager打交道。

在Windows系统下，OS将文件系统模块的各种功能接口打包为一个抽象的System Service，比如对于文件系统打包之后就是File Service，对于设备管理就是Device Service，还有比如Memory Management Service等。被打包之后的Service放到IO Manager头顶，任何程序都可以按照这些Service提供的API来进行相关的调用，比如打开一个文件的操作，Open()。所有的System Service调用动作会被传递给IO Manager进行处理。我们来看一张图，如图19-58所示为Windows系统IO路径简图。

图19-58中一共有10步操作，现一一列举如下。

图19-58 Windows系统IO路径简图

(1) 某时刻，图中的“Subsystem”，这里就是指某个应用程序，向OS (System Service) 发起了对某个文件对象，或者某个设备的Open操作，欲打开这个文件或者设备对其进行进一步的操作。

(2) IO Manager只是一个调度和代理模块，它本身并不知道上层所请求的这个对象（文件或者设备等）到底存放在哪里，是哪一类的对象，文件还是设备等。所以IO Manager向另一个明白人儿，也就是Object Manager发起查询请求以获得结果。同时，IO Manager也会向Securiy模块来查询当前应用程序是否具有访问目标对象的权限。假设本

例中应用程序Open的是比如“D:\Melonhead”文件。

(3) 本例中的目标对象为D分区文件系统下的一个文件“D:\Melonhead”，Object Manager会将这一信息返回给IO Manager（以下简称IOM）。IOM得到这个信息之后首先查看对应的分区或者卷是否已经被挂载，如果尚未挂载，则IOM暂挂上层的Open()操作，转而首先尝试让文件系统挂载这个卷。Windows中存在多种文件系统模块，比如FAT32、NTFS、CDFS等，IOM会依次轮询地尝试让每个文件系统模块来挂载这个卷，直到某个文件系统成功地识别到了卷上的对应的文件系统信息并将其挂载为止。FS挂载之后，IOM继续执行Open()请求。

(4) IOM为这个请求分配对应的内存空间，然后向对应的下层驱动（这里是文件系统驱动，也就是文件系统模块本身）发送一个IRP（IO Request Packet），IRP中包含了上层所请求的所有信息以及完成这个请求所要涉及的底层所有驱动链（文件系统驱动、磁盘设备驱动等）的对应信息。关于IRP的具体结构以及具体流程见下文。

(5) IOM将生成的IRP首先发送给驱动链的顶层第一个驱动，这里就是文件系统本身。文件系统收到IRP之后，会读取IRP中给自己的信息，提取出操作对象和操作内容，然后开始操作。如果应用调用时未指定DIO模式，则文件系统当然首先要查询是否请求的对象数据已经位于Cache中（本例中应用请求Open()一个文件，所以这里的目标数据就是指这个文件的Metadata，FS要读入这个文件的Metadata后才可以响应上层的Open()操作），如果目标数据恰好位于Cache中，那么文件系统直接完成这个IRP请求，将携带有IO完成标志的IRP返回给IOM宣告完成，IOM从而也返回给应用程序宣告完成。如果Cache未命中，则FS需要从底层的磁盘来读取目标数据，这就需要FS在这个IRP中对应的底层设备驱动信息中填入底层设备所要执行的动作，比如读取某某LBA段的内容

等，然后将组装好的新IRP通过调用IOM提供的功能API（详见下文）来将它发送至位于FS下层的设备驱动处。

（6）设备驱动接收到IRP之后，与文件系统做相同的动作，也是首先从IRP中找出给自己的信息，提取出操作对象和内容，然后操控物理设备执行对应的扇区读写。

（7）当驱动链中所有驱动都完成各自的任务之后，底层驱动将带有完成状态标记的IRP返回给IOM。

（8）IOM收到这个IRP之后，检查其状态标志，如果是成功完成，则将会通知上层应用程序打开成功，并且返回给程序一个针对这个文件对象的File Handle。

（9）IOM将这个IRP清除。

（10）IOM返回一个File Handle给程序，以后这个程序就需要使用这个Handle来对这个文件进行其他操作，比如写、读、删等等，其对应的IO请求执行过程会依照上述步骤进行。

可以看到，在OS内核中，每个IO请求都是以IRP为载体来传递的。与外部网络上的数据包相同，IRP包只不过是在内存中传递而不是线缆上。

如图19-59所示为IRP包的数据结构，其中右侧部分为一个“IO Stack Location（IOSL）”数组的放大结构。IOM在发送IRP给驱动链的顶层驱动之前一定要知道完成上层的IO需要的全部底层角色，比如FS、Volume、Device Driver等，这些角色按照发生作用的先后顺序一次排列

图19-59 IRP结构

形成一个自上而下的驱动链，IRP在被IOM初次组装时，会被IOM填充入一个或者多个IOSL。驱动链中有几层驱动，IRP中就有几层IOSL，每层驱动都对应自身的IOSL。

IOSL中包含了本层驱动应该做的事情的描述，比如操作的目标、何种操作等。IOSL中有多项结构用来对其进行描述，比如IRP_MJ_CREATE（打开）、IRP_MJ_READ（读取）、IRP_MJ_WRITE（写入）、IRP_MJ_DEVICE_CONTROL（控制指令）、IRP_MJ_CLOSE（关闭）等。其中MJ表示Major，就是说这些操作都是本层驱动的基本功能，这些基本功能在图中表示为“IRP_MJ_XXX”。还有一类设备自定义的Minor功能，都集合在IRP_MN_XXX中。图中的arguments表示操作的对象，比如所要操作的数据位于何处，长度是多少等。PtrDeviceObject表示本层驱动对应的设备对象指针，PtrFileObject表示待操作的文件对象指针，这个项目只对文件系统驱动层有用处，其他层一般不使用这一项。

在图中左侧，我们可以看到这个IRP中包含上下两个IOSL，上层IOSL已经填充了FS驱动的相关操作信息，下层IOSL尚未空。说明这个IRP的驱动链只有两层，顶层即文件系统驱动（FileSystemDriver，FSD），底层则还尚未被填充，暗示底层驱动一定是存储设备驱动，也同时暗示这个IRP一定是由IOM发送给FSD的但是FSD尚未做出处理。

被IOM组装好的IRP初次被发送到驱动链顶层（本例中是FS）驱动模块时，只有顶层的IOSL被IOM填入，其他层次的IOSL为空，因为此时只有IOM知道应用程序要做什么，而且IOM也不知道FS下层的磁盘驱动要操作哪些扇区。

顶层驱动模块按照IOM填入的IOSL信息执行对应的操作，如果发现

本层驱动的操作必须转交给下层驱动来完成的话（比如系统内核缓存未命中必须从磁盘来读数据），那么本层驱动在执行对应的映射计算后（比如FS将文件offset映射为磁盘扇区段），将下层驱动所要执行的动作信息（比如“读入LBA0~1024的数据”）调用IOM提供的IoAllocateIrp功能函数填入下层IOSL中（由于当前驱动并不会自动得知下层驱动的IOSL位于整个IRP中的位置，所以需要调用IOM提供的IoGetNextIrpStackLocation功能函数来获取）。

填好之后，调用IOM提供的IoCallDrive功能函数将填好的IRP发送给下层驱动（磁盘设备驱动），磁盘设备驱动收到IRP之后，从IRP中读取本层对应的IOSL，也就是刚才由FS驱动填入的IOSL，执行其中的操作。由于磁盘设备驱动为最底层的驱动，所以在它执行完毕后会调用IOM提供的IoCompletionRequest功能函数来将IRP返回给IOM，此时的IRP已经被最底层的驱动在IO Status Block中填入了IO执行状态（成功、错误等）。

IOM收到IRP之后，将最底层对应的IOSL清空，并将IRP再次发送给上一层驱动（本例只有两层，如果有多层则一层一层回传）来让上层驱动判断底层执行的结果是否合格。本例中上层FS驱动是最顶层驱动，所以它会执行IO总完成过程（此处并非IoCompletionRequest），同样在IO Status Block中设置状态然后将IRP返回给IOM，如果状态为成功，则本次IO成功执行，如果状态为其他，则相应触发其他动作。如图19-60所示为Windows系统内IO Manager与驱动层交互的一个示意图，其中包含了IRP流动的路线以及IO流程。可以根据上面的描述来对应图中的步骤。

图19-60 IRP流动过程

1. IOM对Buffered IO处理过程

下面给出一个Windows系统下IO Manager处理Cached/Buffered IO过程示例。如图19-61所示为IOM处理Buffered IO时的流程示意图，其中共有6处关键点，下面就分别描述一下这6处关键点。

图19-61 IOM处理Buffered IO过程示意图

(1) 用户程序将自身数据缓存放置在所分配的虚拟内存空间内某段逻辑上连续的地址之内，这段虚拟内存地址实际上可以被映射到物理内存中并不连续的地址段中，这是任何操作系统Memory Manager都会做的。

(2) 用户程序在向OS发起IO请求时会待写入数据存放的数据缓存地址（对于写请求）或者用于接收数据的缓存地址通知给OS。

(3) OS使用ExAllocatePoolWithTag功能函数在内核的一个内存空间池内创建一个与用户程序对应缓存相同大小的连续的SystemBuffer。

(4) 然后IOM立即向下发起IRP，在IRP中IOM会将SystemBuffer的指针告知顶层驱动程序。

(5) 对于读请求，底层驱动在成功完成数据读入之后，会将逐步读出的数据放置到方才被通知的SystemBuffer地址段中存放，当整个IO完成后，驱动会通知IOM完成信号。当然，底层驱动也可能会有自己的缓存，比如文件系统驱动，文件系统会根据自己的缓存策略来处理这个IO请求，比如，如果Cache Hit，则直接将数据复制到SystemBuffer。对于写请求，IOM会首先将用户程序缓存中待写入的数据复制到方才分配的SystemBuffer中，并立即返回给用户程序完成信号，然后向下发起IRP

过程，下层驱动会直接从SystemBuffer中读数据从而写入底层设备。当然对于文件系统驱动这一层，由于使用了Buffered IO，FS会缓存这个写IO并做优化处理，在Flush被触发后，才会将数据继续向下从驱动写入。

（6）当内核读IO完成后，OS会将SystemBuffer中的内容复制到用户程序缓存空间；当内核写IO完成后，用户程序不会有感知，因为Buffered IO模式下，在用户程序写IO发起后便立即被通知完成了。IO完成后，内核使用ExFreePool功能函数来释放针对本次IO分配的SystemBuffer。每次IO均要分配对应的SystemBuffer。

注意：在上述过程中，当SystemBuffer被分配之后，对应的用户程序Buffer便可以被OS使用换页机制将其内容复制到硬盘上的Pagefile中存放，以便腾出物理内存空间给其他程序使用。如果IO吞吐量很大，但同时依然使用了Buffered IO模式，那么不但因为多次内存数据复制而效率低下，而且会由于在内核分配SystemBuffer时可能由于内存池中沒有对应这么大数据空间的连续内存地址段而造成内核极力去释放内存空间，而首当其冲的就是释放文件系统缓存空间。而释放FS缓存时就又涉及到Flush IO的操作，这些操作在底层又随时可能导致路径中其他模块需要分配内存，而这就成了一个死循环，可能导致系统失去响应。所以IO路径中分配内存的操作需慎之又慎。

2. IOM对DIO处理过程

如图19-62所示为IOM处理DIO过程（底层设备使用DMA方式）的示意图。其中共有6个主要步骤，下面我们就分项描述这6个步骤。

图19-62 IOM处理DIO过程示意图

(1) 用户程序自身的缓存位于操作系统虚拟出来的虚拟内存空间，并且是连续的。这块虚拟连续的空间实际上是被映射到物理内存中的，并且可能是不连续的。操作系统会针对这个程序的缓存空间生成一份Memory Description List (MDL) 用来描述虚拟地址和物理地址的映射关系等信息。

(2) 程序向操作系统发起DIO读请求，并在自身的缓存内预留好了接收读出数据的空间，并将这个空间的地址信息通告给操作系统。

(3) IOM根据收到的缓存地址信息，调用MmProbeAndLockPages功能函数将对应的物理内存地址锁定。

(4) IOM生成IRP以便下发给驱动链的顶层驱动，并在其中给出用于接收本次IO数据的物理内存空间位于MDL中的指针。直到本次IO完成，否则指针对应的内存空间一直处于锁定状态。

(5) 底层设备使用MmGetMdlVirtualAddress功能函数来获取这些指针。

(6) 底层设备从介质中读出数据之后，将数据直接送至这些指针指定的物理内存空间。（由于Windows下DIO模式要求用户程序发起的IO Size必须与最底层设备的最小存储单位对应，所以此处可以直接由底层设备将数据返回而不再途径文件系统返回。但是对于一些底层加入虚拟化效果的驱动，比如卷镜像、RAID等，则最底层设备驱动需要将数据返回给其上层驱动，然后由上层驱动负责将数据置入用户程序缓存。）

综上所述，IOM统管着OS内核中所有驱动以及上层调用接口，为整个系统IO请求流动的骨架。当一个IO进入IOM之后，第一次转发是IOM主动发向驱动链顶层驱动的，随后的进一步下发是由各层驱动主动发起的，此时IO请求流动的驱动力为IoCallDrive函数（IOM提供），各层驱动调用这个函数即可将IRP下发给下层驱动。当然也可以认为上层驱动执行完IRP后将IRP通过IoCallDrive函数扔给IOM，然后IOM再发送给下层驱动，因为IoCallDrive是IOM自身的功能，所以本质上这么理解也可以。

3. 成也Page Cache，败也Page Cache

Page Cache是操作系统内最后一层Cache，再往下走，就是实实在在的物理硬件设备了。在Linux系统中，早在2.2内核版本的时代，除了Page Cache（Pcache）之外，还有一个Buffer Cache（Bcache）处于Pcache下层。这两者的区别是，Pcache用于缓存文件系统层面的文件块数据，而Bcache则用于缓存块设备层面的设备块数据。这两个Cache之间保持严格同步，即一个文件块若处于Pcache中，则这个文件块所对应的磁盘块也必须同时处于Bcache中，针对任何一个块进行的改动，必须也体现到另一个Cache对应的块中，即同步读入同步修改。但是到了Linux 2.6内核之后，Bcache的作用被弱化了，基本上没有用处，文件和块设备的缓存都使用Pcache并且不再同步。

- Pcache与文件或者块设备其实是一种映射关系，即内核会将文件或者块设备上对应的块映射到Pcache中对应的Page，Page作为文件或者块设备上对应块的顶头缓存而存在。在非DIO模式下，针对某个文件或者块设备的IO操作，必须经过Pcache。比如读请求，内核首先检查Pcache中对应这个IO所请求的块的Page中是否

有数据，如果有，则直接返回，无须从底层磁盘读取；对于写请求，内核也是首先将数据写入对应的Page，然后再批量写入磁盘。

- Pcache向下映射到文件块或者块设备块，向上则可以映射到用户程序缓存，这样，利用Pcache作为中间人，程序只要访问自身缓存内映射过来的Pcache对应的Page，就等于访问到了Pcache下所映射的文件块或者块设备块，途中如果遇到Cache非命中的情况，则会产生Page Fault，触发从底层磁盘设备的读操作。这种Pcache上行的映射称为Memory Mapping，Mmapping需要用户程序显式地向操作系统申请，使用mmap()功能函数。利用mmap机制，如果有多个程序共同读取同一个底层块，那么内核无须将这个块分别复制到每个程序自身缓存内，只需要放置一份在Pcache对应的Page中即可，Pcache的上行map会将这个实际的Page分别map到各个程序自身的缓存内，这样就大大降低了物理内存耗费。对于写操作，也同样会写到Pcache中对应的Page，每个程序任何时刻都看到的是相同内容并且最新的Page，也就是说任何时刻只有Pcache中的一份Page数据，这样就可以保证数据的一致性。当需要将被修改的Page写入磁盘时，程序可以调用msync()功能函数进行。

一个文件或者块设备往往比较大，不可能全被映射到Pcache中。但是对于用户程序来讲，其所被分配的内存地址都是虚拟的，往往可以很大，甚至大过物理内存，所以程序向OS内核所申请映射的整个文件/设备或者其一部分，对于虚拟内存来讲是有空间映射的。

由于内核对程序缓存做了超供处理，比如虚拟内存为10MB空间，而实际上内核只给了其5MB的物理内存空间，所以只能拆东墙补西

墙。所以内核需要维护一个Page Table来记录当前实际的虚拟地址与物理Pcache地址的映射关系。

当程序需要对虚拟地址空间内某个page进行IO操作时，内核检查PageTable，如果没有找到这个虚拟Page对应的物理Page，那么内核只能先把当前被占用的物理Page中最少被程序访问的那些想办法腾出来给新请求使用。

腾空间的机制有多种，比如最划算的是先删掉物理Page中那些自从被读入之后就没有改过的，也就是对应的块与下层磁盘上的块内容一致的那些Page，它们占在这里除了增加一些缓存命中率之外别无它用，所以毫不犹豫地将它们标记为Free即可。

其次就是Flush Dirty Page，即如果物理Page中有些已经修改过的但尚未被写入磁盘的Page，那么它们占在这基本上为了那微乎其微的缓存命中率，此时也没多大意义，因为此时此刻燃眉之急是物理空间不够用，所以即刻触发Flush将其写入磁盘后也标记为Free。

更糟糕的是，如果上面两个动作都做完了，物理空间还是不够用，那么还有最后一根救命稻草，即将当前占用的物理Page中的内容写到磁盘上一个特定的空间（Swap分区）或者文件（Pagefile）中暂存，以腾出空间，并且在Page Table中做好记录，这个过程称为Page Out。待什么时候程序又需要访问已经被Page Out的Page了，内核的Cache Manager发现目标Page并没有在物理内存中，此时就算发生了一次Page Fault，那么Cache Manager就得根据Page Table中的记录，从Swap分区或者Pagefile中将对应的之前Page Out出去的Page再读入物理内存供使用，这个过程叫做Page In。

然而，最糟糕的是，物理Page已经全满，该做的也做了，而此时又恰恰需要Page In，那么没办法，只能将物理Page中最少被访问到的（使用LRU算法）Page写到Pagefile或者Swap分区，再将Pagefile或Swap分区中要Page In的Page读入，这种情况称为Page Exchange，是最糟糕的状态。

按理说Pcache的存在会大大提高IO性能，但这只是在一定条件下而已。Pcache以4KB的Page为单位进行读写，只能一次读入或者写入一个或者多个Page，而不可能读写诸如200B，1000B这种除不尽4KB的数据。然而，程序在访问文件或者块设备的时候，却可以以任何offset为起始，发起任何长度的IO请求。这就使得Pcache很难办，一旦遇到这种尺寸不能对齐Page边界而且长度又不能被4KB除尽的IO请求，就会发生本章前文所描述的各种花样的读写惩罚。

另外，4KB的Page Size为x86平台普遍使用，其他一些平台比如安腾，其Page Size为64KB。IO Size与Page Size相差越大，惩罚比例就越大。但是Page Size越小，那么Page Table中所需要的条目就越多，由于Page Table也存在于内存中，所以需要浪费更多的物理内存。

所以说，成也Pcache，败也Pcache。Pcache的成败，最终决定因素还是在用户程序的IO设计，是否充分考虑了各种情况。

另外，对于一些慢速与高速存储介质共存的系统内，比如SSD与机械硬盘共存，此时可以针对SSD设备使用DIO模式，将有限的RAM一级缓存留给更需要它的机械硬盘。

4. Cache、Buffer、Queue的作用层次与关系

从本章一开始给出的那张系统全景IO路径图中可以看到，系统中的各处都会有缓存，还有Queue，同时还存在一种图中尚未画出的东西，就是Buffer。这里有必要讲解一下这三者的层次与关系。

Cache是一个海洋，其中可以有长时间固定不动的东西。而Queue，很显然它就像一个水渠，其中的数据是无时无刻不在流动的。而Buffer更像是一个溢出的水渠，比水渠的空间更大一些，但是Buffer，更应该被翻译为“缓冲”而不是缓存，也就是说Buffer中的数据会很快被清走，或者传输到上游模块，亦或是被传输到下游模块。Buffer有两个作用：一是针对慢速设备提供一个速度适配的缓冲空间，二是为了给某种计算操作提供数据暂存的空间。比如某种存储设备，数据在被读出或者写入介质之前需要进行某种检查或者运算处理，而这种运算是非常底层而不是高层的，那么就有必要为这个设备创建一块缓冲，也就是Buffer空间用于存放计算之前的数据以及运算之后的结果。刻录光盘时系统分配的内存空间视为缓冲，因为这个Buffer中的数据不可能长期被放在里面不动。但是Cache，也就是“缓存”，是比Queue与Buffer更高层的东西，其作用主要是用来加速数据IO而不是缓冲IO流动的，其中可以存在长期不流动的数据。

Cache、Buffer、Queue，这三者的数据流动性是一级比一级高，路径中作用层次一级比一级低，所发挥的功能一级比一级低级。越往宏观速度越慢而空间越大，越往微观速度越快而空间越小，计算机IO路径也完全符合这个自然法则。

19.1.5 底层设备驱动层

1. Windows系统下的驱动链

IO请求从应用程序发起，经历了文件系统、卷管理系统，现在这个IO请求要经历它离开主机操作系统前的最后一关：设备驱动程序。我们以Windows系统的驱动层次架构为例说明操作系统是如何使用底层硬件设备的。

设备分为虚拟设备和实际物理设备，我们在此不讨论虚拟设备。一个实际的物理设备，一般是以某种总线来连接到计算机主板上的，比如PCIX、PCIE总线，主板上的导线连入对应的总线控制器芯片（一般是与北桥或者南桥或者CPU集成的），操作系统就是从总线控制芯片来获取所有外部设备信息的，任何发向或者收自外部设备的数据和控制信息流，都经过这个总线控制器，总线控制器只负责将数据流转发到对应总线的对应地址上的设备，但是数据流本身的流动是由设备DMA部分直接负责的。

所以，操作系统要想看到一个设备，必须通过这个设备所在的总线控制器。操作系统内有一类称为“总线驱动”的驱动程序，这个驱动程序位于操作系统最底层，它可以从总线控制芯片来获取当前总线上的任何动静以及任何已经接入的设备信息，也就是说这个驱动是用来驱动总线控制器的，也叫做“总线控制器驱动”。任何新接入的设备都会在硬件上被总线控制器感受到，从而也就被总线驱动所探知到，然而，总线驱动只能够探知到总线上新接入了一个设备，却不知道这是什么设备，该怎么用。要想使用这个设备，就得在其上再加载一层驱动，也就是这个设备自身的驱动。

比如，如果这个设备是一张PCIE接口的SCSI卡，那么操作系统会首先通过PCIE总线驱动探知到有一个类型为SCSI Host的设备，然后系统会立即将SCSI Port/Miniport驱动加载到这个设备之上，从而可以正常操作SCSI卡。SCSI卡其实本质上也是一个总线控制器，它后端控制着

一个或者多个SCSI总线，在前端则作为PCIX/PCIE等主机总线上的被动设备而存在，所以，SCSI卡的驱动程序也相当于SCSI控制器的总线驱动。所以，加载了这个驱动之后，操作系统就会看到这张SCSI卡上面所连接的SCSI总线信息以及SCSI总线所连接的SCSI设备。然而，此时操作系统只能够看到一个或者多个SCSI设备，却不知道它们各自是什么设备，怎么用它，所以，还需要再加载一层驱动，也就是，PCIE主机总线上的SCSI控制卡上的SCSI总线上的SCSI设备的驱动。

还没结束。每个SCSI设备上还可以存在多个LUN逻辑单元，所以，还需要针对每个LUN加一层驱动。但是一般来讲，这一层驱动与前一层是同一个驱动类型的不同例程罢了，因为一个SCSI Target上的LUN其功能一般与Target本身相同，不会存在某个SCSI Target多个LUN里有的是磁盘而有的却是磁带或者光驱的。至此我们数一数这个驱动链里的驱动数目：主机总线驱动→SCSI控制器驱动→SCSI Target设备驱动→LUN设备驱动。

还没结束。假设某个LUN是一个磁盘设备，至此，操作系统已经知道如何使用这块磁盘了，可以向其中读写数据了。但是磁盘还可以分区，操作系统会对待每个分区就像一块物理磁盘一样。所以，还需要为每个分区再加载一层驱动，这个驱动与上一层驱动其实是一样的，如果使用某种卷管理软件来生成卷而不是分区，则就需要加载卷驱动，这个驱动由卷管理软件提供，作为一种Filter驱动（见下文）类型存在。

至此，还没结束。有了卷或者分区就完结了么？非也。文件系统还没进来呢。文件系统也算是一种驱动，内核为每个卷或者分区加载对应的文件系统驱动之后，整个驱动链才算最终完结，此时应用程序就可以读写文件了。当然如果不使用文件系统，那么可以把驱动链终止在卷一层，甚至LUN一层，应用程序可以通过内核来直接访问到底层设备，比

如对块设备或者裸设备的IO操作。

如图19-63所示，左边为Windows下的一张PCI接口的SCSI卡的驱动链，右侧为一个PCI接口的IDE卡的驱动链。

图19-63 Windows下的驱动链示意图

对于左侧的SCSI驱动链，系统是按照如下过程操作的。

（1）由于系统中有多条PCI总线，总线驱动通过总线控制器发现所有总线之后，会为每个PCI总线创建一个PDO（Physical Device Object）对象，并且再创建一个FDO（Function Device Object）对象表示这个设备（此处为PCI总线）具体功能将由总线驱动来负责。图中的最底层即为其中一个PCI总线的FDO。

（2）总线驱动扫描这个总线上的所有设备，感知到了两个PCI设备，一个IEEE 1394适配卡（在此不介绍）和一个SCSI适配卡。所以总线驱动为这两个设备分别创建各自的PDO，由于总线驱动不知道如何使用这两个设备，所以不能为其创建FDO。

（3）以SCSI适配卡为例，内核将SCSI Port/Miniport驱动，也就是SCSI控制器驱动加载到其PDO上，并同时生成一个针对这个设备的FDO，即功能性对象，意即加载了SCSI控制器驱动后这个设备自身的功能就可以被操作系统使用了。SCSI控制器驱动，也就是Port/Miniport驱动被加载之后，就可以从这个SCSI卡后端所连接的SCSI总线感知到具体的SCSI Target设备了。本例中SCSI总线上连接了一块磁盘和一个CDROM光驱，所以驱动为这两个设备创建了各自的PDO。由于SCSI控制器驱动只负责SCSI总线的维护以及向SCSI Target设备发送和接收数

据，并不知道如何使用这些SCSI设备，也并不知道向其发送的数据的具体内容和格式，所以不能为这些设备创建FDO，这个工作将由更上层驱动完成。

（4）在CDROM设备的PDO之上，系统加载了一个Audio Filter Driver，即音频过滤驱动。过滤驱动作为一个夹层角色存在，可以插入任意两层常规驱动之间而不改变夹着它的上下两层驱动间的接口。关于过滤驱动将在下文介绍。

（5）此时操作系统已经发现了一块硬盘和一个CDROM光驱，为了使用这两个设备，操作系统必须要加载针对它们的功能性驱动。针对磁盘设备，内核加载了Disk Class Driver；针对CDROM光驱，内核加载了CDROM Class Driver。Class Driver的意思即“类驱动”，也就是说针对任何磁盘设备，只要是磁盘，不管是哪个厂家生产的什么规格的磁盘，统统属于磁盘类，它们都可以使用同一个驱动程序来行使它们的功能，除非有一些厂商自行设计的特殊功能，则需要加载对应的Miniclass驱动。类似的Class Driver还有诸如CDROM、磁带机、机械手、显卡、键盘、鼠标等。

（6）至此，操作系统已经可以使用对应的设备了，比如读写等操作。但是硬盘是可以分区的，针对每一个分区，内核还会再次加载一层Disk Class Driver。

（7）最后，本例中还包括了一个磁盘加密过滤驱动，这个驱动的作用是在文件系统驱动（本例中未表示，位于磁盘加密过滤驱动之上）和Disk Class Driver之间将自身插入，截取文件系统本应该直接发向磁盘驱动的IO请求，加密其中的数据，然后将加密的数据除了内容之外，原封不动地发送给磁盘驱动。现在很多的加密程序都是使用底层过滤驱动

来实现的。

对于IDE适配卡的驱动链加载过程可以参考SCSI适配卡的过程，二者大同小异，这里不再描述。

2. Windows系统下的IO设备驱动类型

Windows系统将IO设备的驱动程序按照层次分为了两大类，即上层的Class Driver以及下层的Port Driver。关于这两种驱动程序的角色和作用，上文已经做了简要介绍。这里将进一步介绍这两种驱动的具体功能项目。

如图19-64所示为这两种驱动的层次示意图。先来看左侧部分，应用程序想要执行某些IO操作，它可以调用Window提供的API，比如打开某个文件读写等，Window API会自动向下调用对应的内核层驱动比如文件系统驱动来执行对应的功能；应用程序也可以调用用户态驱动程序（在用户态执行的驱动程序）来执行一些高层的、可以在用户态就完成的功能，如果不能在用户态完成，那么用户态驱动会自行调用Windows API执行对应操作。一些打印机驱动程序就属于用户态驱动程序。

图19-64 设备驱动层次和类型

内核为多数的设备类型都提供了Class Driver来实现基本功能，比如鼠标，不管鼠标有几个其他按键，其基本的功能无非就是左右键和滚轮，Class Driver就提供了对这些功能的支持。至于鼠标上的其他按键以及其他一些更高级的功能，可以由厂商提供各自的Miniclass Driver或者Filter Driver来单独实现这些功能。Class与Miniclass互相链接以共同完成设备的全部功能。Port Driver则是专门用来驱动各种外部总线控制器

的，比如USB、IDE、SCSI等；同样，Miniport Driver则是用来驱动这些控制器上的一些厂商特定的功能的，不同厂商可能有不同的特殊功能，所以不同厂商需要有不同的Miniport Driver。最底层则是主机内部总线驱动，比如PCIE总线驱动。

再来看右侧，这个图表示IO在这些驱动层次之间的流动。上层将IRP（见本章前文）发向下层的Storage Class Driver，Class Driver将IRP转换为SRB（SCSI Request Block），SRB中包含了CDB（Command Description Block）以及其他一些内容（有人可能不解，Storage Class Driver对任何设备都使用SCSI协议集么？底层如果是ATA磁盘的话怎么办呢？是的，这个问题我们下文再描述），Port Driver接收到SRB之后，对SRB进行分析，并将SRB转换为外部总线所要求的指令格式发送给对应的设备。

下面我们对每层Driver进行分别介绍。

3. Storage Class Driver

Class Driver是驱动某个最终设备的功能性驱动。Windows针对所有的硬盘设备，不管其是SCSI还是IDE或者U盘或者SSD，亦或是磁盘阵列的LUN、卷等，都使用相同的驱动程序；同样，针对其他大类的设备，也都使用相同的驱动。所以称其为“Class Driver”。Storage Class Driver专指存储类的驱动，比如Disk Class Driver、CDROM Class Driver、Tape Class Driver等。Class Driver由操作系统自身提供，不需要厂商针对自己的设备再开发完整的一套Class Driver，如果某个设备的功能并不能由操作系统自带的Class Driver来驱动，那么厂商只需要开发一个附属在主Class Driver上的额外驱动模块即可。这个额外的小模块就叫做Miniclass Driver，下文中将会描述。

Class Driver处理上层（比如用户程序或者文件系统驱动等）下发的IRP，根据IRP的请求内容，将请求转换成包含有SCSI CDB的SCSI Request Block（SRB），然后将SRB发向下层驱动，比如SCSI Port Driver或者位于其下的过滤驱动（如果有的话）。Storage Class Driver统一使用SCSI指令集来操作底层设备，如果底层物理设备并非SCSI的，比如ATA磁盘，那么会由位于Class Driver下层的ATA Port Driver（下文介绍）将SCSI协议映射翻译为ATA协议指令。此外，Class Driver也并不关心底层设备对应的总线物理地址，底层寻址的操作由Port Driver执行。

Storage Class Driver需要执行的具体功能如下。

- 为每个具体设备生成FDO对象。如果设备上存在分区，则对每个分区生成PDO和FDO。
- 获取每个设备的属性信息，比如是否支持Write Cache及其模式、最大传输单位等。
- 处理上层下发的IRP请求，将IRP映射翻译成SRB并继续下发。
- 维护每个请求的超时机制。
- 根据底层反馈的所能接受的每个请求的最大传输单元，将上层的IRP请求的数据长度分割为底层所能接受的值。
- 错误处理机制，处理底层所不能处理的错误，比如SCSI Check-Condition等。

1) SCSI PassThrough支持

Storage Class Driver支持SCSI PassThrough，即用户程序可以直接生成CDB而发向底层设备，Class Driver收到上层的CDB之后，会做一定的格式检查，判断为正确格式之后，还需要检查IO Size，如果大于底层规

定的最大值，则做分割操作，然后直接将CDB封装到SRB中发送给Port Driver处理。

2) 关于Command Queue

对于Class Driver层，没有必要对所有上层发来的IRP做Queue处理，因为Queue的处理由其下层的Port Driver全权负责，Class Driver只管将上层的IRP翻译成SRB然后源源不断地向Port Driver的Queue里填充即可。Port Driver会为每个LUN分别创建Queue，比如SCSI Tagged Queue。当某个LUN的IO发生错误时，Port Driver会将对应的Queue进行Freeze，并将自己无法处理的错误报告给Class Driver，由后者进行处理。Class Driver会查询底层设备所支持的Queue方式，如果支持Queue，则Class Driver会在每个SRB中设置对应的标记告诉Port Driver本次IO可以使用对应的Queue，并且还附带了对本次IO进行SCSI Command Queue的策略信息，比如：SRB_SIMPLE_TAG_REQUEST（SCSI控制器可以将这个IO重排到任何位置）、SRB_HEAD_OF_QUEUE_TAG_REQUEST（控制器必须将这个IO放到队列首处，首先执行）或者SRB_ORDERED_QUEUE_TAG_REQUEST（SCSI控制器必须将这个IO放到队列尾部按照顺序执行）。

3) 关于获取设备属性信息

Class Driver需要获取它所驱动的设备（此处其实是获取SCSI或者IDE、1394等控制器和具体的SCSI/IDE设备的属性信息，因为Class Driver下层是Port Driver，Class Driver发出的SRB必须服从Port Driver的要求，也就是服从SCSI/IDE控制器的要求，同时也要符合最终设备的要求）的属性信息，这些属性信息包括：

- SCSI/IDE控制器的最大传输单元，即每次IO的最大可读写的扇区长度；
- 是否SCSI/IDE控制器在DMA时可以读写不连续的物理内存Page，数量是多少；
- SCSI/IDE控制器对Buffer边界对齐的要求信息；
- SCSI/IDE控制器是否支持SCSI的TCQ以及是否支持基于每个LUN的TCQ；

SCSI/IDE控制器是否支持WriteBack模式的写缓存，具体类型是什么，比如是否有电池保护等。

4) 关于设备写缓存的模式

一些外部设备比如RAID适配器、SCSI适配器，都具有自己的缓存。这些缓存如果用来做预读操作，那么没有任何问题，但是如果用Write Back的模式来缓存写数据，那么在有些适配卡没有电池保护其缓存的情况下，一旦发生供电故障或者Down机，则被缓存的写数据就会永久丢失。Class Driver层有必要知道底层设备是否支持Write Back缓存，如果支持，是否有电池保护等信息。对于没有电池保护的WriteBack模式的设备缓存，Class Driver有两种解决办法。第一种是使用SCSI SYNCHRONIZE CACHE指令来强制让设备将缓存内的所有Dirty写数据写入存储介质中，这种方法需要消耗很多的设备自身的处理资源，因为设备一旦接收到这个指令，就需要全力以赴地执行。而写盘动作是一个慢速动作，在执行这个动作期间，新进入的写IO可能得不到处理，而上层在此时的反应就仿佛设备挂死一般。如果频繁地使用SCSI SYNCHRONIZE CACHE指令，系统整体性能将会非常差。第二种方法则是针对每个写IO使用Write Through标记（FUA置1，即Force Unit Access），这样，设备在从总线上接收这个IO请求写的的数据入缓存的同

时，将数据写向介质中，只有成功写入了介质，才返回完成信号，而写入完成后，缓存中的这些数据将被视作预读内容处理。使用Write Through的方法虽然也不能提高写性能，但是其相对SCSI SYNCHRONIZE CACHE指令的方式来讲节约了设备处理资源。有些设备并不支持FUA，但是它对待每个写IO都会做Write Through处理，也就是原生的WT模式写缓存。而有些设备则带有电池为其缓存供电，那么这时候，Class Driver就没有必要发起Write Through或者SCSI SYNCHRONIZE CACHE了。

Class Driver使用IOCTL_STORAGE_QUERY_PROPERTY功能函数来向Port Driver查询设备属性信息，其中就有关于写缓存的信息，具体写缓存信息如下。

- 设备是否具有写缓存。
- 设备具有何种写缓存类型。又包含两种具体类型：Write Back和原生Write Through。
- 设备是否支持SCSI SYNCHRONIZE CACHE指令。
- 设备是否具有电池保护。

提示：我们在Linux系统启动的时候经常会看到一些信息，比如在发现某个sd设备的时候，后面会跟着一条描述信息，比如“Write Through”，这就表示Linux下的块设备驱动探寻到了具体设备所支持的写缓存方式。

5) 关于Class Driver层的请求重试

大部分的IO请求重试都是由Port Driver来做的，比如数据校验错误、仲裁超时、Target设备繁忙、总线Reset等底层错误。针对这些底层

错误的重试，需要由Port Driver负责，上层的Class Driver完全不知道也不关心，如果Port Driver重试多次后依然无法成功，从而向Class Driver报告，那么此时Class Driver不应该再次重试以做无用功。然而，一旦遇到一些上层逻辑的错误，比如Check Conditions等，那么此时就需要Class Driver介入，解决对应的错误并且重试之前的请求，由于错误发生之后，原本积压在Port Driver的Queue队列中的所有IO请求都将被冻住，等待错误恢复，所以Class Driver需要将重试的请求标记SRB_HEAD_OF_QUEUE_TAG

_REQUEST以便Port Driver优先执行这个IO请求。

4. Storage Miniclass Driver

上文中曾经简要描述过，Storage Miniclass Driver的角色类似一种Agent Driver，用来针对特殊的设备实现对应的特殊功能。这些功能由于不是普遍的每个厂家都来实现，所以在Class Driver中并不包含，厂商自行开发Miniclass Driver然后将其与主Class Driver链接起来形成一个驱动对而存在。

5. Storage Port Driver

Port Driver是位于Class Driver之下的一层驱动程序，同样也是由Windows操作系统自身提供。Port Driver是用来驱动各种外部总线控制器的，而外部总线控制器一般又位于主机总线之上，也就是表现为一块接入主机PCIE等总线的适配卡，所以说Port Driver又可以被看做是这些适配卡的驱动程序。如果这些适配卡是专门用于适配与存储有关的总线或者设备的话，那么就将这些Port Driver称为Storage Port Driver。

所有类型的基于SCSI的存储适配器，不管它是SCSI卡、FC卡还是iSCSI卡，亦或是RAID卡，它们对外提供的总是一个或者多个SCSI Target设备以及LUN，上层的驱动比如Class Driver不管这些LUN是通过IP网络达到、通过FC网络达到还是通过SCSI总线网络达到，统一对待。这也就是“适配器”三个字的本质所在，适配的就是这种不同的后端网络传输方式。既然这些适配器都对外具有统一性，那么它们的驱动程序就一定具有统一性，这个被统一的驱动程序就是Windows自带的Storage Port Driver。Port Driver将大部分统一的功能驱动编写好，任何适配器厂商只需要编写实现自己特殊功能的那部分Miniport Driver即可，大大简化了复杂度以及工作量。

适配器驱动与适配器硬件的工作分工是不同的。比如并行SCSI适配器和它对应的Port Driver之间，适配器上的SCSI控制器硬件本身所做的工作包括SCSI总线的初始化扫描、数据传输时的总线仲裁等，而其对应的Port Driver所做的工作则是将IO请求提交给控制器，控制器再将请求发送给对应的SCSI Target，当Target执行完毕返回ACK信号给控制器之后，控制器将这个信号也发送给Port Driver表示成功执行，如果Target执行过程中出现错误，那么控制器会将错误信息传递给Port Driver处理。对于FC适配器，与并行SCSI控制器所不同的地方就是由并行SCSI变为了FC网络和链路，所以一切FC网络和链路层的逻辑都需要由FC控制器硬件来完成，而网络链路所承载的SCSI上层逻辑依然由其对应的Port Driver来处理，Port Driver使用SCSI上层逻辑来收发数据和控制Target，与并行SCSI适配器所不同的只是FC卡的Port Driver可以探测并且控制一些FC网络和链路层的参数，比如设置WWN或者更改FC Port类型等控制性操作。同理，对于RAID适配器等也都是这个道理。

在Windows中提供三种Storage Port Driver，分别是：SCSI Port

Driver（SCSI PORT.sys文件）、ATA Port Driver（ataport.sys文件）以及 Storport Driver（storport.sys文件）。下面分别介绍。

1) SCSI Port Driver

SCSI Port Driver是一个最标准、最传统的Windows下的用于驱动基于SCSI协议的适配器的Port Driver，在Windows 2003之后的版本，SCSI Port Driver被Storport Driver（下文介绍）取代，后者专门针对高性能、高带宽存储网络适配器做了优化操作。但是Windows 2003系统中依然可以使用SCSI Port Driver。

SCSI Port Driver主要提供如下功能。

- 提供其驱动的适配器的各种属性信息给上层的Class Driver，比如最大传输单元的限制以及Write Cache属性信息等。Class Driver主动向Port Driver（SCSI PORT Driver）发送 IOCTL_STORAGE_QUERY_PROPERTY 查询请求，Port Driver则将所有信息放入STORAGE_ADAPTER_DESCRIPTOR中返回给Class Driver。
- 确保适配卡所连接的所有设备处于正常状态，确保尚未加电的设备不被上层逻辑所使用。
- 负责提取并分析Class Driver发送的SRB中所包含的CDB，并将CDB封装为对应物理设备可识别的指令。
- 由于Class Driver自身并没有请求队列，针对每个IRP请求都是尽力而为地转换为SRB而直接充入Port Driver的Queue中，这个过程是一个异步过程，Class Driver无须等待上一个请求的完成信号就可以发送下一个请求。Port Driver处必须维护请求队列，因为底层物理设备的处理能力是有限的，而且可能随时发生错误，当物

理设备处理不过来而处于Busy状态时，Port Driver必须等待并且重试；当发生底层错误时，同样也需要重试。另外，磁盘是支持Queue的，比如FC和SAS所支持的TCQ以及SATA所支持的NCQ，Queue的效果可参考本章后面章节。这一系列的逻辑都需要Queue的存在。

- 管理自身的请求队列。比如，当发生上层逻辑错误（比如Check Conditions）时，Port Driver自行将对应设备的Queue冻结住并将错误返回给Class Driver进行处理，等待处理结果返回来之后，Port Driver才继续执行Queue中的请求。为何非要冻结呢？一个IO出错，并不影响下一个IO的执行。是的，但是第一，IO之间是有上层逻辑关联的，比如当前发生错误的IO为针对LBA1的写入请求，而队列后紧跟着这个IO之后的是针对LBA1的读请求，如果在这个写IO发生错误之后继续执行读IO，那么读取到的就是时序上不一致的数据；第二，当前IO发生上层逻辑错误，那么就意味着很有可能后续的IO都会发生错误，Class Driver很可能会Cancel掉所有受影响的IO请求，此时如果Port Driver擅自处理，也是不允许的，所以，错误之后的Queue冻结是有必要的。
- 除了自身管理自身的队列之外，Port Driver还提供接口供上层逻辑来控制其自身的Queue。Class Driver以及其他上层逻辑可以向Port Driver发送Lock、Unlock、Freeze、Unfreeze请求以操纵其队列。比如上层使用SRB_FUNCTION_RELEASE_QUEUE功能调用来Unfreeze已经被冻结的Queue。
- 当发生底层错误时，对IO请求进行重试操作。底层错误指诸如数据校验错误、传输时错误、目标设备繁忙、仲裁冲突等。此外，Class Driver会为每个IO设定Timeout计时器。但是Port Driver对其有更灵活的处理，比如在执行IO过程中遇到链路中断，那么Port Driver此时可能会暂停Timeout计时器，当链路恢复时再继续计

时。

- 当发生上层逻辑错误时，将错误用SCSI-2 Sense Status格式封装并传递给Class Driver，Class Driver根据Sense返回值做出相应处理。
- 提供接口支持Miniport Driver。

（1）关于SCSI Port Driver与Class Driver之间的接口

在前文描述Class Driver时曾经给出了一些这二者之间应该互相沟通的东西，比如探询设备属性、SCSI Pass Through支持、分割IO请求、传输SRB、错误处理等。由于Class Driver统一使用SCSI协议指令集，而SCSI Port Driver也是专门驱动基于SCSI协议的存储适配器的驱动，所以二者之间的CDB信息是完全一致的。SCSI Port Driver在收到Class Driver下发的SRB之后，只需要将其中包含的CDB加入一些诸如端口号、路径、目标Target ID、LUN ID等寻址信息然后透传至SCSI Miniport Driver，Miniport Driver再传递给适配卡上的控制器即可。如果是ATA Port Driver，那么Port Driver必须将基于SCSI的CDB转换映射成对应的ATA指令从而发送给IDE控制器。

当出现上层逻辑错误时比如Check Conditions，Class Driver会要求Port Driver向控制器发起Request Sense请求，Port Driver接收到要求之后就会生成一个Request Sense指令并将其发送给SCSI控制器，SCSI控制器再根据接收到的CDB中的目标设备地址和ID信息，将对应的请求通过对应的总线发送给目标设备，目标设备返回的Sense数据中包含了具体的错误代码，Port Driver再将收到的Sense数据返回给Class Driver处理。

提示： SCSI控制器不是一个存储目标设备，它只是为所有SCSI总线上的目标设备提供一种数据传输控制的一个关口设

备，是一个“总线控制器”，它并不感知具体的请求内容，只管从正确的目标收发数据。所以驱动程序必须在IO请求中附带有目标设备的地址信息。

（2）关于SCSI Port Driver与SCSI Miniport Driver之间的接口

二者之间通过一系列具体的功能函数以及回调等来实现沟通，具体细节不做介绍。如果底层的SCSI适配器支持Tagged Command Queue（TCQ），那么Port Driver会将自身Queue中的请求异步地批量发送给Miniport Driver，Miniport Driver也同样会异步地将请求通过硬件总线传递给SCSI适配卡；如果适配卡不支持任何Queue技术，那么Port Driver就会同步地将请求发送给Miniport Driver，IO请求一个接一个地执行。同样，在目标设备与SCSI控制器之间也是这样做的。

（3）关于SCSI Port Driver的队列管理

当出现底层错误时，SCSI Port Driver将对应目标设备的Queue冻结以等待Class Driver做出处理。当Class Driver需要对目标设备做一些影响较大的设置更改时，比如修改目标设备的电源状态，会要求Port Driver将Queue锁住不再执行，然后将对应的修改设置的请求发送到Port Driver，这个请求中带有Bypass Queue的标记，所以此时虽然Queue被锁住，但是Port Driver根据这个标记依然会将请求发送到对应的目标设备，完成之后，Class Driver会对Queue进行Unlock以恢复正常执行状态。上层驱动比如Class Driver可以在下发的SRB中将SRB_FUNCTION_FLUSH_QUEUE置位以要求Port Driver对其Queue中的所有请求进行Flush操作，Port Driver接收到这个请求之后，会将当前Queue中的所有请求执行完毕，并且还命令SCSI控制器或者RAID卡将它们自身的Cache中的Dirty Data写入磁盘。

2) ATA Port Driver

在Windows NT 4.0时代，ATA设备的驱动是作为一个附属于SCSI Port Driver的Miniport Driver实现的（atapi.sys文件），这个Miniport Driver负责将SRB转换为ATA协议指令。到了Win 2000和Win XP时代，这个Miniport Driver脱离了SCSI Port Driver，并且被分离成为两个层次，位于Class Driver之下的一层为IDE Port Driver（atapi.sys文件），专门负责管理IDE控制器的不同IDE通道以及翻译SRB为ATA指令，其再之下的一层为IDE Controller Driver（pciide.sys文件）/Minidriver（pciide.sys文件）对，专门负责驱动IDE控制器硬件。到了Windows Vista及之后的时代，ATA设备驱动又被做了一些更改，这里就不介绍了。

3) Storport Driver

Storport Driver（storport.sys文件）是Windows 2003及以后的系统中用来替代SCSI Port Driver的。Storport Driver在性能和接口上做了一些改进，更加适合于诸如RAID卡以及FC卡等高性能高带宽的存储IO适配器。Storport Driver相对SCSI Port Driver在接口方面的改动很小，所以，厂商自开发的SCSI Miniport Driver升级到Storport Miniport Driver也是比较容易的。

Storport Driver相对于SCSI Port Driver的最大改进就是Storport Driver与Storport Miniport Driver之间的接口由原来的同步变为了异步，即使物理设备不支持Queue。而SCSI Port Driver只有在物理设备支持Queue的时候才异步地向Miniport Driver发送IO请求。其次，Storport Miniport Driver不再需要在中断与IO发起之间保持同步，二者可以各干各的，所以类似一种全双工模式。Storport Port Driver使用Push方式将IO强行推送

给Miniport Driver，而SCSI Port Driver则靠Miniport Driver每执行完一个IO后从Port Driver的Queue中将IO请求Pull过来执行。

另外，Storport Driver在Queue管理方面也有提升，Storport Driver为每个LUN维护一个Queue，Queue最大深度为254，即Queue中可以排有254个尚待处理的IO请求，如图19-65所示。

图19-65 Storport对Queue的处理

另外，当出现目标设备繁忙的错误时，SCSI Miniport Driver不会向SCSI Port Driver报告这个状态而只会自行重试发送IO请求，Storport Driver则允许Storport Miniport Driver报告Busy状态，从而Storport Driver会暂停向Miniport Driver发送IO请求。

使用Storport Driver有一些其他限制，这里就不再介绍了。

4) Storage Miniport Driver

在Windows系统下，我们常见的RAID卡、FC卡、SCSI卡、iSCSI卡的驱动程序其实几乎都是Miniport这种架构，这些卡件的功能区别体现在其后端所连接的总线方式。比如SCSI连接的就是最原始的并行SCSI链路，而FC卡则是连接了FC网络，iSCSI当然就是连接了IP网络了，至于RAID卡，其实只是在上面罗列的任何一种卡上附带有RAID控制器而已，加了一层虚拟化操作。前文曾经说过，这些适配器本是同根生，其驱动也具有统一性，但是每种设备不可能都一样，总有一些区别于其他设备的细化功能。而这些功能上的区别，到了软件层面就体现在Miniport Driver上，而位于Miniport Driver上面的所有软件层次模块看到的其实都是同样的东西。

Miniport Driver表现为一个DLL文件，它与Port Driver进行动态链接。针对每种Port Driver，Miniport Driver又可以分为SCSI Miniport Driver、Storport Miniport Driver和ATA Port Miniport Driver。

Miniport Driver需要执行的部分任务包括：

- 发现适配器并探寻其各种属性；
- 初始化适配器；
- 执行数据IO过程；
- 处理中断、总线Reset等事件；
- 维护超时计时器。

5) Storage Filter Driver

过滤驱动泛指插入某两个驱动层次之间，用自己设定的处理过程来改变原有的IO内容或者路径或者丢弃某些符合设定条件的IO请求的一种特殊驱动程序。过滤驱动由于插入在两层原有驱动之间，所以它必须适配上下层，让上下层都感觉不到它的存在。

典型的例子，比如键盘上的一些自定义的功能键、按键组合等；或者文件加密工具，它嵌入Class Driver之上和Filesystem Driver之下来发挥作用；再比如卷镜像软件，它插入Port Driver之上和Class Driver之下；还有存储系统中常用的多路径软件，也是一种过滤驱动。

6. Linux下的存储系统驱动链

1) SCSI设备驱动链

上文介绍了Windows下的Storage Driver层次架构，现在介绍一下

Linux下的层次架构。其实Linux下的存储系统驱动程序链与Windows下的主体层次大致是一致的，只是并没有Windows下那么简洁的层次而已。如图19-66所示为Linux下的一个典型的驱动链层次。

图19-66 Linux下的SCSI设备驱动链

位于首层的当然是UNIX类系统的著名的VFS目录层，这个就不用多做解释了。

下一层是Solid File System层，其中Solid为作者自行选择的用词，之所以使用Solid这个词是为了显示出与VFS目录层的本质区别。Solid FS层为实，表现为阴的实质基础，VFS层为虚，表现为阳的生发能量，所以用Solid表示“阴实”。Solid FS层可以是任何一种实质的文件系统比如EXT2/3或者NTFS等。

再下一层，就是系统缓存了，全局的Page Cache缓存区，可Prefetch也可以Write Back。再下一层是Generic Block Layer，这一层是FS抽象的结束，FS将文件IO映射为Block IO之后，就要发送到Generic Block Layer。

再往下就是各个块设备的驱动层，通用块层将对应的IO请求发送给对应的块设备驱动，著名的IO Scheduler调度器就运行在此层，此处调度器需要一个Queue。

再往下是一个公共服务层，即SCSI Middle Layer，这一层的功能是专门处理SCSI上层逻辑，比如数据IO、错误处理等机制，还负责将上层的块IO映射翻译成对应的SCSI协议CDB，然后发向其下层的SCSI Host Controller，即SCSI控制器驱动，这里也需要有一个Queue。到了这里，再往下就是系统总线和外部总线、适配卡等硬件了。

如果非要将Linux下驱动链与Windows下的做类比的话，那么Generic Block Layer和Block Device Driver Layer以及SCSI Middle Layer这三者合起来提供的功能相当于Windows下的SCSI Class Driver；SCSI Host Controller Driver就相当于Windows下的SCSI Port Driver。Windows下的存储子系统驱动链中只有一个Queue，即Port Driver所维护的Queue，而Linux下有两处。

下面详细介绍一下一个IO请求在Linux内核中的流动过程。

(1) 用户程序对一个VFS目录对象（比如/mnt/test.file）发起Open()操作。

(2) 随后进行read()操作，内核接收到read()请求之后，首先进入了文件系统的处理流程当中，文件系统入口为generic_file_read()。入口函数判断本次IO是否为Direct IO，如果是则调用generic_file_direct_IO()函数来进入DIO流程，如果是Buffered IO，则调用do_generic_file_read()函数进入Cache Hit流程。do_generic_file_read()其实只是一个外壳包装，其内部其实是do_generic_mapping_read()。

(3) 文件的区段都会被映射到Page Cache中对应的Page中，这个函数首先检查请求的数据对应的Page是否已有数据填充，如果有，则Cache Hit，直接将其中数据返回给上层；如果未命中，则调用readpage()函数来向底层发起一个块请求，readpage()调用了mpage_readpage()。

(4) mpage_readpage()又调用了do_mpage_readpage()函数来创建一个bio请求（即Block IO，这个bio相当于Windows系统下的IRP）。

(5) 然后接着调用mpage_bio_submit()处理这个bio请求，

mpage_bio_submit()其实调用了submit_bio()函数将bio发送给下层函数generic_make_request()。generic_make_request()函数为通用块层的入口函数，这个函数调用make_request_fn()函数将bio发送给下层，也就是Block Device Driver层的IO Scheduler的队列中。

(6) make_request_fn()函数为块设备驱动层的入口函数，也是IO Scheduler的入口函数，这个函数包装了__make_request()函数，它是实现IO Scheduler的调度功能的主要函数。关于IO Scheduler的具体功能将在下文介绍。

(7) IO Scheduler通过调用request_fn()函数从而将bio下发给了SCSI控制器驱动，SCSI控制器驱动程序调用相关函数将这个bio转换成SCSI指令，然后调用scsi_dispatch_cmd()将指令发送给对应的SCSI控制器硬件。

以上过程可以对应图19-67所示的流程图来理解。

图19-67 Linux系统块设备IO流程图

2) Linux对ATA设备的驱动链

如图19-68所示，与Windows下的ATA Port Driver类似，Linux也使用一种LibATA库来负责将SCSI协议转换为ATA协议并发送给ATA控制器驱动程序。

图19-68 Linux 下ATA 设备驱动链

3) IO Scheduler

IO Scheduler是Linux下专用来对IO进行优化的一个模块，所有针对

底层存储设备的IO都要经过这个模块的优化操作，然后将被优化之后的IO顺序地放到底层存储控制器驱动程序的Queue中，如图19-69所示。

图19-69 IO 调度器基本架构图

IO Scheduler首先要做的优化是对所有Block IO进行重新排序操作，即Reorder，让它们按照LBA地址进行排序。因为磁盘的磁头臂每执行一个IO就至少需要一次寻道操作，如果将上层无序的IO直接发给磁盘执行，那么磁头臂可能会做很多不必要的来回摆动，极大影响IO性能；而如果将IO预先按照LBA地址排序然后再发给磁盘，那么磁头臂就可以持续地向一个方向寻道，避免了摆回去再摆过来的浪费。

其次，IO Scheduler还可以对相邻LBA地址的同类型的IO请求进行合并操作，即Merge。比如有两个读IO先后到达，第一个的起始LBA地址为0，长度1024，第二个的起始LBA为1024，长度3072，则IO Scheduler就可以将这两个IO合并成一个起始地址为LBA0、长度为4096的大读IO。当然，如果后到的同类型的IO的目标地址与之前的某个IO地址完全重合，则调度器会直接抛弃之前的IO，如果后到的同类型的IO与之前的IO目标地址有交集但是不全重合，那么调度器会取并集。

另外，既然IO Scheduler对IO进行了有区别的对待，那么它就一定要在区别的同时保证每个IO的利益，比如有某个IO其目的LBA地址是一个在一段时间内比较冷门的地方，一般IO没去那里的，所以IO Scheduler想找几个IO和它做伴，以减少运费，但是苦耐一直没找到，那么只能将它放在队列里等着，但是过了很长时间依然也没去执行它。这种情况IO Scheduler需要尽量避免。

还会发生另外一种情况，即有些IO拉帮结派，专横跋扈得很，动辄

一大批一下子进来排队，弄的队列里被它们占了大部分地方。比如在程序使用Buffered IO模式+同步IO调用的时候，对于写操作的执行总比读操作快，因为写操作永远都是被内核接受并且数据被复制到内核缓存后就返回成功信号；而读则不行，读操作在缓存未命中的情况下，必须从存储介质中读出并且返回给程序才算完成，而这种情况下读操作是很慢的，在同步IO调用情况下，读操作只能一个接一个的来，而写则可以大批地执行。其次，读总比写要重要，这里的重要是对于程序自身来讲的，因为只要一个程序发起读操作，就证明它需要这份数据才能够做出下一步动作，如果迟迟不给它这份数据，那么它很有可能就处于等待状态，处理效率和吞吐量就降低了。但是程序如果发起写操作，证明它已经处理完了这些数据需要将它们存到硬盘，而如果迟迟不处理这些写数据，那么程序的效率和计算吞吐量可能根本不会受到影响或者受到很小的影响，程序会继续读入数据、处理计算，然后尝试写入硬盘，虽然此时写入的时候有积压。IO Scheduler必须解决这个问题，即“读饿死”问题。

同时，在多个用户进程同时发起IO时，不管是读还是写，也存在争抢问题，有些进程发起大量的IO，而有些则发起很少量的IO，此时一样会出现队列争抢问题。IO Scheduler有必要本着公平公正的原则处理。

如图19-70所示为Linux下的IO Scheduler对IO进行Merge操作时的基本流程。其中FrontMerge和BackMerge分开对待是因为Merge之后必然需要Split，调度器根据Front和Back来判断被Merge的IO是追加在之前IO的后面还是被贴附到之前IO的前面，这样就可以在Split的时候根据Front或Back来进行Split操作。

图19-70 IO调度器对IO进行Merge时的流程图

在Linux 2.6内核中，有4种IO调度方式：NOOP、Deadline、Anticipatory、CFQ。每种方式的侧重点不同，实现方式也不同，但是基本的Merge操作，这4种调度方式都会执行。这些调度方式可以在系统运行的时候任意更改，并不会影响IO的执行。下面简要介绍一下这4种调度方式。

（1）NOOP调度方式

NOOP调度方式可以说并没有去调度什么，而只是将所有IO按照先进先出的顺序进行排队，它只执行Merge操作，而不去改变IO的顺序以适应磁头臂寻道。如果底层存储介质为机械硬盘，并且IO属性为随机IO的话，那么使用这种方式无异于自找苦吃。这种调度方式可以说是专门为以Flash为介质的存储系统而生的，因为Flash介质的存储系统不需要机械寻道，所以是否重排IO对其没有任何意义；但是Merge IO依然具有意义，Merge IO任何情况下都具有意义。不对IO进行重排，带来的好处是极大地节省了系统资源，包括内存和CPU资源。

（2）Deadline调度方式

Deadline调度方式会对Queue中的IO进行Merge和重排操作。它会创建两个队列和两个表，一个队列是按照读IO的LBA起始地址进行排序的读IO队列，与其对应的有一个按照IO进入顺序排列的FIFO表，用来记录队列中所有IO的进入顺序；同样，对于写IO操作也维护这样一个队列和一个表。常规情况下，调度器将IO重排后下发给下层驱动执行，但是这样势必会引起上文中提到的IO被冷落的情况，所以Deadline调度器会维护计时器，当FIFO表中有某个IO的进入时间戳与当前时间相差到一定程度时，也就是说这个IO在队列中待了太久也没被执行，那么调度器会抛开其他一切因素将这个IO下发到下层驱动。这也是其被称为

Deadline调度器的原因。Deadline调度器针对读IO的停留阈值时间为500ms，而对于写IO则为5s，这样考虑的原因是因为上文中提到过的“读饿死”的情况。如图19-71所示为Deadline调度器的队列构成以及调用的相关函数。

图19-71 Deadline调度器队列结构

（3）Anticipatory调度方式

顾名思义，Anticipatory调度器会做某种预测从而执行对应的调度算法。它会根据队列中已经存在的IO的起始地址信息来预测是否下一个到来的IO其起始地址与队列中的某个IO会相近甚至相邻。如果它判断的结果为是，那么它会等待一段时间，等待符合条件的IO进入。如果进入相邻的IO，那么就将其与现存的IO进行Merge操作；如果是相近的IO，则可以将其重排，磁头臂不需要远距离寻道。利用这种方法来最大限度地提高效率。这种预测仅限于读IO，对于写IO不做预测。

如图19-72所示为Anticipatory调度器的等待判断流程。其中，Mean thinktime表示自从上一个读IO完成之后到接收到写一个读IO的相隔时间的平均值；Mean seekdistance表示前一个读IO的结束地址与后一个读IO的起始地址之间的距离平均值。

图19-72 Anticipatory调度器等待判断流程

Anticipatory调度器如此的预测和等待，为的只是多拉快跑，这样做的根本原因还是因为运力不够，比如底层存储系统是很慢的机械硬盘等情况。

（4）CFQ调度方式

CFQ全称为Completely Fair Queue，完全彻底的公平的队列调度。CFQ调度模式除了具备基本的Merge、重排等功能之外，还体现了绝对公平。这里的公平二字，体现在对多个进程，或者多个进程组、多个不同用户各自的所有进程、多个不同用户组各自的所有进程所发出的IO，按照这些单位作为调度粒度，在这些单位之间保持完全公平。比如两个用户分别各自运行了两个进程，那么CFQ调度器会为每个用户的两个进程创建对应的队列，在这两个队列之间保持公平，比如从队列1被提取下发了4个IO，那么会转到队列2提取下发4个IO。如图19-73所示为单进程调度粒度的CFQ调度器的队列结构。

对前文所说的“读饿死”情况，CFQ使用了一些参数来控制这种情况的发生。另外，CFQ对Synchronous IO的处理优先级要比Asynchronous IO高很多，因为同步IO表示程序此时是阻塞的，程序在等待返回，而异步IO则表明程序没有在等待，CFQ给予同步IO以高优先级是理所当然的。

图19-73 CFQ调度器队列结构

CFQ中的一些可调节参数如下。

- **back_seek_max** 这个参数规定了后向寻道距离的最大值，单位用KB表示，默认值为16384KB。这个参数效仿于Anticipatory调度模式下的mean seekdistance参数，只不过CFQ中这个参数是一个定值，而Anticipatory调度模式下的这个值等于历史统计结果平均值，但是参数决定的行为对于两个调度模式下是不同的。Queue中将要被执行的IO（此时Queue中的IO已经按照目标地址和磁头臂摆动的方向进行插入排序了）的目标地址如果位于当前磁头臂摆动的方向后方（即需要摆回去才能执行这个IO），而且

相隔当前的磁头位置（也就是上一个IO的结束地址）的距离大于这个参数给定的值，那么CFQ会判断为后向寻道距离过长，不划算，所以继续执行队列中的正方向的IO，因为磁头向一个方向连续摆动时的寻道开销是最少的。如果距离小于这个值，那么CFQ认为摆回去是划算的，执行当前IO。

- **back_seek_penalty** 这个参数的作用与back_seek_max类似，只不过更加纯粹一些。它同样决定了磁头臂后向回摆是否划算。如果当前磁头位置位于a处，而Queue中有两个IO，目标位置分别为后向a-n，和前向a+m，如果 $(a-n) < (a+m) / \text{back_seek_penalty}$ ，则CFQ将后向的IO与前向的IO同等对待，先摆回磁头臂执行后向IO，再执行前向IO。
- **fifo_expire_async** 这个参数规定了Queue中的某个异步IO如果因为重排等因素而一直没有被执行从而超过了这个参数所规定的时间的话，那么会强制执行这个IO。默认值为250ms。
- **fifo_expire_sync** 这个参数与fifo_expire_async类似，只不过是针对Queue中的同步IO，默认值为125ms。可以看到CFQ是偏向优先执行同步IO的。
- **slice_sync** CFQ之所以公平，是因为它会轮流从每个进程、进程组、用户或用户组的Queue中提取IO并下发给设备驱动。为了保证公平，CFQ对每个Queue的执行时间是分片的，这个值就规定了CFQ对每个同步IO的Queue所执行的时间，默认为100ms。
- **slice_async** 与slice_sync类似。默认值40ms。
- **slice_async_rq** 这个值限制了CFQ每次执行异步IO队列时最大可以提取下发的IO请求数量，默认为2。当被下发了对应数值的IO之后，CFQ将转到下一个Queue继续执行。对于同步IO队列，没有这个参数的限制，可见CFQ对同步IO大开绿灯。
- **slice_idle** 对于同步IO队列，如果轮到某个同步IO队列被执行

时，队列中却是空的，那么CFQ将等待slice_idle所规定的时间；如果超过这个时间仍未有IO进入，那么CFQ转向下一个Queue执行。slice_idle默认值为8ms。对于异步IO队列，CFQ并不等待。可见CFQ对同步IO的重视程度。

- **Quantum** 这个值规定了底层存储控制器驱动的Queue Depth，CFQ不能将多于这个值所规定的IO数量发送给底层设备驱动。

提示： 随着外部存储系统越来越智能化，这些IO优化任务其中大部分完全可以交给外部存储控制器来执行，主机端完全没有必要多此一举，浪费系统资源，所以目前IO Scheduler正在被逐渐弱化，仅仅对于本地硬盘或者JBOD阵列有些许效果。

因此对于使用外部智能磁盘阵列存储系统的Linux系统，可以使用NOOP或者CFQ模式的Scheduler。由于外部存储控制器无法感知到进入的IO所附属的主机端进程，因此不可能做到向主机端IO Scheduler平衡多进程IO，所以CFQ是有必要使用的。

主机SCSI或者FC控制器驱动程序都会探知到底层硬件是否支持TCQ或者NCQ，如果支持，则驱动程序会异步地将批量IO充入底层的Queue；如果不支持，则只会一个一个地将IO下发，严重影响性能。不支持Queue的一般是单个低端磁盘，而磁盘同一时刻只能执行一个IO操作，所以控制器感知到这一点，才会一个一个地将IO下发。而对于外部磁盘阵列系统，在接收到主机端设备属性查询时都会返回支持Queue以及Battery Backup Write Back Cache。

另外，不光外部只能用盘阵，对于支持TCQ的SCSI磁盘或者支持NCQ的SATA磁盘，如果对应的控制器硬件以及驱动程序也同时支持Queue的话，那么IO Scheduler的效果同样也会被弱化，此时选择NOOP或者CFQ方式即可。

对于Linux操作系统，调度器的队列深度设置，可以通过编辑/sys/block/sda/queue/nr_requests文件中对应的值来实现。队列深度越大，那么IO优化的效果越明显，但是每个IO的延迟也会随之升高。关于Queue深度与IO延迟的关系见下文描述。

(5) IO Scheduler对IO的Merge实例

程序对块设备做IO时，诸如O_SYNC（类似WRITE_THROUGH，并不指同步IO调用）等参数依然适用，并且由于块设备的写惩罚特性，在UNIX系统下，使用O_SYNC模式调用时，写惩罚最严重，如果不使用O_SYNC或者WRITE_THROUGH之类的参数，那么写数据会被块设备驱动进行Merge操作。下面这个测试使用的命令为dd if=/dev/zero of=/dev/sdb bs=512 count=1000，操作很快完成，使用iostat工具检测到了这个过程中的所有IO情况。

Device:	rrqm/s	wrqm/s	r/s	w/s	rsec/s	wsec/s	av
sda	0.00	0.00	0.00	0.00	0.00	0.00	0.00
sdb	0.00	115.00	125.00	10.00	1000.00	1000.00	14.82 0.0
wrqm/s	The number of write requests merged per second that w						
r/s	The number of read requests that were issued to the						
w/s	The number of write requests that were issued to the						
wsec/s	The number sector transferred to the device per senco						

- ① dd发起“offset0 length 512B”的写IO到内核，由于使用了Buffered IO，内核立即通知dd完成了。
- ② 内核首先读出这个块设备在Page Cache中映射的逻辑上的第一个4KB的Page也就是Page0，然后修改前512B，再将修改后的Page0放入块

设备Queue中等待写入。

③ 在上一个Page0尚未写入时，dd完全可能再次发起“offset=512 length=512”的下一个写请求，同样，立即被告知完成。

④ 内核为了服务第二次512B的写入，需要再次读入Page0。由于之前的Dirty Page0还在Page Cache中，所以本次读会命中缓存，内核将本次的Page0的对应区域修改之后，再次将这个Page0放入块设备Queue中等待写入。

⑤ 块设备的IO Scheduler前后两次收到了两个针对同一个目标地址段的写IO，根据调度规则，当然是将前一个写IO抛弃，只保留后一个，或者也可以认为后者覆盖了前者。

⑥ 程序发起第3次IO、第4次IO，内核依然是读缓存命中，同样，写入的Page0也会将前一个覆盖掉。

⑦ 当程序发起第5个IO时，内核必须要读出Page1了，然后继续重复与前面相同的过程。

这样算来，平均每8次程序发起的IO才会触发一次真正的对一个Page的读和写操作。那么程序共执行了1000次IO， $1000 \div 8 = 125$ ，根据测试的结果输出，也确实符合了这个数值。但是写操作却为10次，另外的115次被Merge了。也就是说，Page0和Page1或者更多的相邻Page被Merge成一次大的IO写操作了。

（6）IO Scheduler的副作用

是药三分毒。IO调度器在对IO进行了优化的同时，也带来了严重的副作用，这种副作用是任何层面的IO优化Queue所无法避免的，包括

TCQ、NCQ。

比如某程序打开一个块设备/dev/sda进行IO操作，程序使用DIO+异步IO调用。T1时刻程序针对某地址做了固定长度的读操作，假设缓存未命中，内核将这个读BIO放入块设备驱动的IO Scheduler队列中；但是紧接着T2时刻，程序又发起了针对同一地址的固定长度的写操作，假设内核将这个写IO也放入了IO Scheduler的Queue中，但是经过调度之后，这个写操作被提前了，提到了读的前面，先执行，那么这个读操作就不会读到被这个写所覆盖之前的内容，而读到的是写覆盖之后的内容。同样，对于先写后读，如果读被提前，那么读到的也是过期的数据。

如果使用Buffered IO，则先写后读的情况下，对于后来的读会产生缓存命中，内核根本不会将BIO发向块设备，也就不会带来时序问题。如果是先读后写而且读未命中的情况，那么可能会发生读数据未返回之前、写数据已经在缓存里呆着了，而且也有可能发生写数据从缓存中被flush刷盘，进入IO Scheduler，此时读请求依然未被执行，还在IO Scheduler中，那么这个写依然可能会被提到读前面，当然，等这么久那个“读”还排着队的几率是很低的。

如果是多进程共同IO同一个目标，这种问题一样会发生，不过可以对目标进行Exclusive模式的锁定，这样就从根本上杜绝了多进程之间的时序性问题，但是依然存在单进程的时序性问题。如果你真的认为锁定了就没有时序问题了，你就错了。锁定与否与底层的乱序是无关的，即便是程序A锁定文件a进行IO，程序A针对a的IO都会被排队，A解锁之后B继续对a做IO，B的IO也会被排队，而A和B的IO依然会在底层乱序，A和B所看到的内容也可能都是过期的内容。

这个问题是IO Scheduler无法避免的，但是可以从应用程序层去避

免。比如在DIO模式下，对于先读后写的情况，避免发生即可。话说回来，一个程序先发出读，在没收到返回数据的情况下又发出了写去打算覆盖之前的数据，这本身是不是有点逻辑错乱的意思呢？就比如你一开始想拿一个东西，后来想想又不拿了，然后扔了一颗炸弹把东西炸了，但随后你拿到的却不是之前的东西，是一堆被炸碎的东西。也就是说，既然在你没拿到东西之前，你又决定不要了，那么此时这个东西本身就没有意义了，此时你应该知道这个结果，而不是进入脑袋错乱的状态。

而对于先写然后在没返回之前又发出读，一般情况下是没有这个必要的，因为你自己知道之前你写进去什么东西，此时为何还要再把它拿出来看看呢？在这么短的时间内，数据一般都依然在应用的缓存内，没必要向系统发出读请求，可以直接在应用层命中缓存。别说，还真有这么一类患有强迫症的程序，这类程序就是专门干这个的，因为磁盘有时是不可靠的。对于写操作，磁盘返回成功之后，数据未必真的被写入磁盘，甚至对于Write Through标记的IO也是如此，也就是说磁盘并没有将数据成功地写入盘片，但是依然向上层返回了成功应答，这种极端发生的情况称为“Lost Write”。这类程序的作用就是为了探察这种情况的发生。对于这种程序，就要注意切不可在刚刚写完之后就立即发起读，一定要过一段时间才发起，以避免在IO调度器的队列里与之前的写IO碰上。而对于使用Buffered IO的程序来说，先写后读没问题（见上文），对于先读后写的状况，与DIO模式下的处理方式相同，即不应该在没有收到读返回数据的情况下接着又发起写IO。

彻底解决这个问题的办法：对于单个线程，同步IO即可；多线程之间或者多个程序之间，无解。

关于**Lost Write**: Lost Write是一种发生概率非常小的事件，但是一旦发生，其所带来的直接后果就是数据不一致。如果受

影响的是文件系统元数据中的关键数据，比如跟入口指针等，那么就有可能直接导致文件系统全面错乱，需要数据恢复。而如果是某应用程序所存储的关键数据，比如账目，那么当应用程序下次读出这份上次尚未被写入成功的数据进行处理之后，就可能发生错账，进而可能引发重大生产责任事故。

如何解决Lost Write？有多种方法，一种最笨最耗资源的方法便是在写入之后，立即再次对这个地址发起读操作（前提是磁盘本身关闭了Write Cache，即使用Write Through模式），将得到的数据与之前的待写入数据进行比对来检查是否上一次真的被写入了，这个动作应该由存储控制器来执行。但是这样做需要对之前写入的数据进行留底，需要对每个块进行Hash计算并保存起来，稍后读出对应地址数据之后再次Hash，比对两次Hash，如果相同则表示确实成功地写入了；如果不同，则表示未被成功写入。此时就需要从RAID组中这个块对应的条带中的其他块来进行XOR运算反算出当时应该被写入的内容。为何可以用条带中其他Segment来反算这块数据呢？因为当时这个条带发生写入的时候，可以认为只有这个块发生了Lost Write，而其他盘同时发生Lost Write的几率微乎其微，所以可以假设其他Segment中数据都是一致的。这种办法虽然笨而且极其耗费资源，但是却可以彻底解决Lost Write。总之一定要确保写入改地址之后，下一个针对这个地址的IO一定要是控制器发起的这个验证读操作，因为如果不立即检查的话，一旦间隙中有针对这个地址的非验证读操作，不管是直接读还是由于更改条带中其他Segment连带导致的对本Segment的读，那么就证明不一致的数据已经对应用系统产生了影响，不管是潜在的还是立即导致后果的。

块设备由于底层IO Scheduler可能造成IO乱序重排执行的情况，在发生系统Down机等情况时，底层数据的一致性就无法得到保证。由于文件系统建立在块设备之上，所以FSCK是一种恢复数据一致性的手段，但是FSCK只能保证文件系统元数据的一致却保证不了数据实际内容的一致性，所以，关键应用程序都直接使用完全透传程序IO请求的字符设备进行IO操作。

7. 系统IO路径中的Queue Depth及Queue Length

在系统IO路径中，有各种各样的Queue，各种层次的Queue，从应用层一直到设备驱动层。对于Queue来说，有两个非常重要的概念：Queue Depth以及Queue Length。乍一看二者似乎意义相同，但是细究起来两者并不一样。前者是指某个Queue的额定深度，即这个Queue最大能充满多少条记录或者多少个IO；后者则指当前这个Queue当前存在的，或者叫做积压的，记录或者IO数量。

随着上层发起的IO请求不断地下发，系统路径下层各处的Queue可能都会面临不同程度的积压，因为底层硬件的处理能力总是有限的，尤其是对于存储系统来讲，在高压下很快达到瓶颈，底层的瓶颈会一层一层地反映到上层。在上层IO压力很小的时候，系统路径上的Queue几乎都没有积压，随着上层IO压力逐渐增加，位于路径中最底层的Queue，也就是设备驱动程序处的Queue会首先出现积压（此处抛开IO Scheduler处的Queue不谈，因为此处的Queue根据一定的调度算法，总会有一定积压）。当积压到一定程度之后，这个Queue就进入了Queue Full的状态，此时，往上一层的Queue开始积压，满了之后，再上一层的Queue就开始积压。就这样一直反映到程序层，此时就会直接感知到性能的急剧下降。如果整个系统中只有一个进程在做同步+DIO模式的

IO操作，那么就可以观察到系统路径的最底层的Queue中几乎任意时刻都是空的，因为此时Queue不可能积压，任意时刻只有1个IO请求在从上流到下，但是我们在探察的时候能碰上Queue中恰好有这个IO存在的时候的几率非常低。当有大量并发的IO被发起的时候，整个所经过的路径中的Queue可能会被充满。

监测到的Queue Length越长，就证明Queue积压越大，那么单个IO的延迟就越高，对应的计算公式为： $IO\ Latency = (Queue\ Length + 1) \times IO\ Service\ Time$ ，其中IO Service Time表示存储系统硬件处理每个IO所耗费的时间，IO Service Time对于存储系统自身来说，也相当于IO Latency，每一层的IO Latency都等于本层的 $(Queue\ Length + 1) \times$ 其下层的IO Latency，所以这个公式可以最终改为： $Layer\ N\ IO\ Latency = (Layer\ N\ Queue\ Length + 1) \times Layer\ (N-1)\ IO\ Latency$ 。根据这个公式，可以推论出：如果存储最底层出现瓶颈，那么最底层的IO Latency（原生Latency）就会升高，它的升高直接导致了其Queue Length越积越多而升高，此二者的升高又成倍地升高了上一层的IO Latency（累积Latency）。一层层的扩大之后，整个系统路径中的Queue积压越严重，那么最终应用程序层的IO延迟就会严重地成倍地升高。公式中的变量为Queue Length而不是Queue Depth，这里一定要注意。

Queue积压到一定程度，导致整个Queue充满溢出之后，下层会向上返回Queue Full通知，上层得知后会做出相应处理，比如降低IO发送速率等。Queue Full的出现表示存储系统瓶颈已经达到了不可调和的地步，这种情况一般不会出现。

主机端底层发生Queue积压情况下系统瓶颈点判断方法：如果链路带宽和外部存储系统的IOPS尚未达到额定饱和值而Queue积压，那么说明瓶颈归于磁盘；如果IOPS饱和，说明瓶颈归于存储控制器的处理能

力；如果带宽饱和，那么瓶颈归于链路速率本身。

当出现Queue积压情况时，主机端的IO延迟会呈指数曲线模式升高，唯一可以做的就是降低主机端的IO并发度或者Queue Depth，这样做只是治标不治本。降低主机端IO并发度或者Queue Depth的本质其实是要应用做出妥协。这里又可以分为两种情况。第一种是底层IO与上层业务响应速度没有直接联系，即底层IO响应变慢不影响实际业务响应时间，那么我们此时就可以大刀阔斧地降低主机端的Queue Depth和IO并发度，减轻存储系统的压力从而也为其他使用存储系统的客户端打通道路。这种情况的一个典型例子就是FS预读，比如上文中论述FS预读时给出的一组数据，在AIX系统下将j2_maxpagereadahead调节成512，也就是让IO Size为2MB大小。当然底层协议本身并不允许这么大的IO Size，所以OS会自动将这个值切开成多个小IO，即每次预读都会使用多个IO来读取数据，不管预读线程是单线程异步IO还是多线程同步IO，单位时间内都会产生大量预读IO从而塞满Queue，造成Queue积压。而此时应用程序的IO响应时间与FS预读力度之间并没有很大的联系，如果此时因为FS的预读造成了Queue积压，那么我们就可以毫不犹豫地将j2_maxpagereadahead降低，比如64。比如下面的数据，将j2_maxpagereadahead调节为512之后，cp操作所耗费的时间并没有相对调节为64时有多大的提升（相反还有所降低），但是反而造成了存储系统的Queue积压，底层的单个IO响应时间急剧从接近0增加到8~9ms。也就是说底层的响应时间增加了，但是其并没有严重影响上层响应时间（慢了不到1s，在测试误差允许范围内）。

```
aix-cn22:/#iioo -o j2_maxPageReadAhead=512
Setting j2_maxPageReadAhead to 512
aix-cn22:/#mount /dev/fslv02 /mnt3
```

```
aix-cn22:/#time cp /mnt3/c /mnt2
```

```
real    0m12.46s
```

```
user    0m0.22s
```

```
sys     0m9.81s
```

Read Ops	Write Ops	Other Ops	QFull	Read kB	Write kB	Average Latency	Queue Length	Lun
363	0	0	0	91136	0	8.69	30.02	/vol/
368	0	0	0	95232	0	9.29	3.05	/vol/
747	0	0	0	79592	0	3.72	3.00	/vol/

```
aix-cn22:/#iio -o j2_maxPageReadAhead=64
```

```
Setting j2_maxPageReadAhead to 64
```

```
aix-cn22:/#mount /dev/fslv02 /mnt3
```

```
aix-cn22:/#time cp /mnt3/c /mnt2
```

```
real    0m11.67s
```

```
user    0m0.22s
```

```
sys     0m9.34s
```

Read Ops	Write Ops	Other Ops	QFull	Read kB	Write kB	Average Latency	Queue Length	Lun
383	0	0	0	97856	0	1.04	0.03	/vol/
437	0	0	0	95936	0	0.90	0.03	/vol/
380	0	0	0	97152	0	1.05	0.03	/vol/

参数的调节需要严格的试探，因为每个系统都不同，IO类型也不同，我们不可能向Nio一样钻到Marix里面去探究代码的执行，但是我们至少可以做试探来探究和想象底层的行为。比如确定j2_maxpagereadahead参数的最适值，就需要我们逐渐增大这个值然后调

查IOPS、带宽和IO响应时间这三者，设置过低则不能发挥性能，设置过高又会顾此失彼，根据需求，总有一个最合适的值存在。

第二种情况则是底层IO的响应时间直接关系到应用程序的响应时间，那么此时我们不能轻易就降低Queue Depth或者并发度，因为这样做虽然可以使存储系统负载减轻一些，但是同样造成了应用程序响应更加缓慢。此时就要综合来决策，查看一下是否其他主机端有可以被砍掉的不必要的Queue Depth和IO并发度，如果有则砍掉以减轻存储压力。如果这样处理之后仍不能改善，那么唯一可以做的就是消除存储端瓶颈，增加磁盘，或者其他手段，比如集群化、迁移数据到高性能设备上，或者使用SSD等。

既然Queue积压是个坏现象，Queue中只有1个甚至0个IO等待才是最理想的状况，那么为何不将Queue Depth调成2算了？这里要理解的是，Queue积压表示某处出现瓶颈。但是在系统仍有充足的处理能力，各处没有产生瓶颈的时候，存储系统对IO的处理是很快的，而如果我们把Queue Depth调成2，那么会喂不饱存储系统。就像吃饭时候一样，你饿了，想吃一大碗米饭，但是只给你盛来一茶碗米饭，你吃的爽么？吃完了虽然可以无限制再去盛一碗再吃，但是这样做效率显然不如一下子盛一大碗来得快。要将Queue Depth调成适当的值是有必要的。当用大碗盛饭之后，你一开始吃得很快，满一碗立即吃空一碗，但是后来你饱了，下饭速度越来越慢，此时碗里剩饭就会越来越多，此时已经达到瓶颈了。同样，也不能把Queue Depth设置的无限大，因为你敢给出这个饭量，证明你对IO的消化能力很棒，那么客户端就会按照这个饭量来给你盛饭；但是你如果没那么大饭量而硬充胖子的话，结果就是造成Queue积压。所以存储系统在设计的时候就会根据自身处理能力和IO类型比率等诸多因素来最终确定一个对外Queue Depth。但是这个Queue

Depth往往也并不能代表真实处理能力，比如对于小块连续并发IO这种最佳IO方式。这就相当于喝稀饭，一气很快就喝完了，此时你不给他一个大碗对不起这速度，但是如果改吃米饭，那一时半会儿吃不完，此时你如果还给他个大碗，那就容易造成积压。所以Queue Depth的确定最终只是一个折中值，设计人性化的存储系统可以让用户自行设置Queue Depth以满足不同需要。

对于一些人为积压的Queue，比如IO Scheduler中的Queue，某种意义上讲，这个Queue必须处于积压状态，因为只有Queue中存有一定数量的IO之后，IO调度算法才会有效果。但是话又说回来，如果底层设备处理能力够强大，速度也够快，快到即使不使用IO调度也能够快速吃掉Queue里的存货，那么此时要IO调度有何用呢？人为积压有何用呢？一点用没有反而影响性能了。

所以，任何时刻只要发现Queue中有存货，那就一定证明底层有瓶颈，存货越多，瓶颈越大。这是铁的事实。系统IO路径内任意两个模块之间的Queue都遵循上述原则，不仅是主机与存储设备之间的Queue，存储系统内部各个模块之间的Queue同样遵守。

如果有多台主机通过同一个端口共同访问一台存储设备，那么此时需要尽量确保每台主机的适配器的Queue Depth都相等，并且确保它们之和等于这台存储设备上一个端口的Queue Depth。这样做，第一可以保证存储系统的Queue不会发生Queue Full的情况，第二可以保证每台主机充入存储设备Queue中的IO数量大致相同，不会饿死任何一台主机。当然也可以为不同的主机设定不同的Queue Depth以便实现性能分级，但是同样要确保所有主机客户端的Queue Depth之和不要超过存储系统上这个端口的Queue Depth。

在某些OS下可以针对每个LUN设置各自的Queue Depth，当然，这样做一定需要对应的存储控制器驱动程序支持。比如在AIX系统下，就可以设置每个hdisk的Queue Depth，只要保证系统中所有hdisk的Queue Depth之和不大于对应的底层控制器驱动提供的总Queue Depth即可，这样就可以针对不同hdisk给予不同的优先级。

8. Queue Length与IOPS的关系

一块15KB RPM的FC或者SAS磁盘，其随机读IOPS大概在±400左右，随机写IOPS要低于读，有些磁盘写与读相差不太大，有些磁盘则相差较大，大概20%。但是达到这个峰值，是需要做出一定妥协的，这个妥协就是IO延迟。但凡要让磁盘达到它最大的IOPS（指随机IOPS而不是连续IOPS，单块磁盘连续读IOPS在IO Size很小时可达上千），就一定要将磁盘的队列充满到一定程度。为何要这样呢？因为磁盘都是支持一定的Queue算法的，比如电梯算法，它会将充入Queue中的所有IO进行重排，以便磁头臂摆向同一个方向时可以捎带执行更多的IO操作。而如果你只向磁盘Queue中充入了1个IO请求，执行完之后再充入1个IO，那么此时执行效率是最低的，但是相对于每个IO来讲，这个IO本身被执行的速度将会是最快的，也就是延迟最低，因为此时磁头臂只服务于这一个IO请求；相反，一次性向Queue中充入多个IO请求，那么磁头臂就会在一次摆动过程中服务于多个IO请求，而这些IO在微观上也是一个一个被执行的，只不过执行效率提高了很多。但是同时，每个IO的响应时间（延迟），也就是磁盘处理每个IO所花费的时间也就提升了（为何？见下文），但是此时磁盘所表现出来的整体IOPS也提升了。乍一看好像有些矛盾，为何延迟升高了，也就是处理每个IO所耗费的时间提升了，而IOPS却不降反升呢？我们就来分析一下这个问题。建议读者在这里先停下自己思考，然后将自己的思考与作者稍后给出的分析做比对，这样

可以加深印象。

如表19-3所示为某块15KB RPM的3Gb/s SAS硬盘的测试结果。测试使用单台PC Server+SAS卡进行，使用IOmeter作为测试工具测裸盘，在IOmeter界面中设置不同的“Outstanding IO”来调查不同Queue Length对磁盘IOPS的影响，IO类型为4KB块的100%随机读。表中ART表示Average Response Time，MRT表示Maximum Reponse Time。

表19-3 SAS硬盘Queue Length与IOPS

当Queue Length（下用QL代替）为1时，属于典型的同步IO调用模式场景。此时系统IO路径中的所有Queue，包括IOmeter应用程序Queue、块设备驱动Queue、HBA适配卡Queue、磁盘Queue中任何时刻只有这一个IO，并且这个IO只出现在这一串Queue中的一个之中。

整条路径只服务于一个IO，此时IOmeter所得出的ART也就约等于底层磁盘的平均寻道时间，本例中也就是5.6ms。

当QL增加到2的时候，系统路径中的所有Queue都不可能产生积压（Queue Depth一般为8以上），一定会“充入便下发”，那么此时这两个IO一定是被HBA直接下发到磁盘Queue中等待执行。

磁盘会使用NCQ、TCQ等算法，结合当前磁头臂的位置，对Queue中的IO做电梯算法来决定先执行哪个后执行哪个，那么后执行的就必须等待先执行的执行完毕之后才能被执行，也就是说IO会在磁盘Queue中发生等待。这样，理论上讲，两个IO执行完所耗费的时间应该等于执行1个IO耗费的平均时间乘以2即可？

不是的，平均寻道时间是使用一个一个的随机IO下发到磁盘而统计

出来的，多个IO经过电梯算法排队之后，会收到一定的优化效果。比如可能向相同方向摆动两次就可以执行完这两个IO，而如果将这两个IO分两次下发，那么磁头可能必须经过向两个方向的摆动寻道才可以完成这两个IO。所以本例中，当 $QL=2$ 时， ART 小于 $5.6 \times 2 = 11.2ms$ 。依此类推，当 QL 越来越长之后，这种效果所获得的收益就越强。

有人会产生疑问： $1000ms/ART (ms) = IOPS$ ，这个等式应当成立，为何按照本例中的结果之后，除了 $QL=1$ 时成立，而其他 QL 值对应的数据被代入后却都不成立呢？比如 $QL=2$ 时， $1000/9.87=101.3$ ，而实际却是202.62，正好是前者的2倍，而 $QL=4$ 时所得到的结果也是这样，实际值为理论值的4倍，可以发现 $IOPS = [1000ms/ART (ms)] \times QL$ 这个规律，而如果将单位 ms 变为标准单位 s ，那么这个公式就变为： $IOPS=QL/ART$ 。读者看到这个公式是否眼熟？对了，它就是本书第3章中所给出的公式。

这个公式本身是正确的，那么问题出在哪里？还得从 $IOPS=QL/ART$ 这个公式的本质入手。它的含义是：每秒IO操作数等于 QL 除以平均响应时间，而按照常理，每秒IO操作数应当等于 $1s$ 除以每个IO的平均响应时间。前者是实测所得，后者是理论所得，似乎都没有错。错在哪？关键就在“平均响应时间”上， ART 到底是指每个IO被执行完成所耗费的时间，还是整个 QL 队列的所有IO均被执行完所耗费的时间。如果是前者，那么公式 $IOPS=1/ART$ 显然是成立的；而如果是后者，那么公式 $IOPS=QL/ART$ 就是成立的。那么也就是说表19-3中的 ART 是指队列中所有IO均被执行完成所耗费的平均（磁盘可以批量返回一大批IO结果，甚至整Queue批量执行和返回，详见下面一节）时间，既然这样， $IOPS$ 也就当然要用 QL 而不是1来除以这个 ART 了。

15KB RPM SAS盘的最高IOPS为400+左右，那么我们用 $1000/400$

=2.5ms，也就是说，NCQ、TCQ等队列优化算法最终可以使得单个IO的执行时间从本例中平均的5.6ms降低到2.5ms，提升了约50%。但是却增加了IO在队列里的等待时间，等待多长时间可以参考上一节中给出的计算方法：IO延迟=IO等待时间+IO执行时间。所以，Queue的存在导致磁盘可以在单位时间内执行更多的IO，也就是IO执行时间确实降低了，但是所有IO的延迟时间却因为IO等待时间的加入而变得很高，每个IO的等待时间等于排在其前面的所有IO的执行时间的累加，可想而知。

Queue的出现，会将每个IO的执行时间降低，低于平均寻道时间（本例中的5.6ms），但是却有一个副作用，也就是造成队列中IO等待的时间加长。因为毕竟IO到了磁头臂这一层一定是一个一个被执行的，一次性充入Queue中多个IO，只是为了在一段时间内能够让磁盘执行更多的IO，但是所付出的代价便是这些IO需要预先排队等待，增加了IO总响应时间，也就是IO延迟。

那么这样做究竟是不是得不偿失呢？谁获益了？谁损失了？可以这么说，在不同层面看这个问题会有不同的结论。先从单块磁盘层面来看，Queue让单块磁盘单位时间内可以执行的IO数量增加了，这当然是好事情，获益于电梯算法，当然是提高了效率，使得单盘IOPS从QL=1时的178提高到了QL=32时的389，同时单个IO的执行时间也降低了。所以，Queue对于磁盘来讲是绝对的全面获益。再看看IO适配器一层，比如SAS适配器驱动程序处，这一处也维护着多个Queue，包括接收、发送、已完成、未完成等Queue，上层下发的IO先充入这一处Queue，然后驱动程序再将Queue中的IO通过SAS网络传送给对应的磁盘Queue中，此时SAS适配器驱动需要等待这些IO完成，而这个等待时间，也就是IO延迟，会随着QL的提高而累加提高，所以，Queue的出现对于SAS适配器来讲有所损失；但是也有受益，即整体IOPS增加。同样，系统

IO路径中再往上，包括块设备驱动、文件系统、内核其他IO模块，一直到最终的应用程序层，皆表现为IO延迟上的损失以及IOPS上的收益。应用层所感知到的IO延迟是其下层所有延迟的累加。只有磁盘是唯一一个全面获益的角色。

但是，刚才说过，在不同层面看问题会有不同的结论。如果从整个系统全局地来看，QL的提高会致使整体系统的IOPS提高，从这一点上来看确实是整体受益的。那么对于应用程序层来讲，它可以选择一个接一个地发送IO，也可以选择批量异步地发送IO。如果业务层决定应用程序就要一个接一个地发送，那么此时情况是从源头就没有那么多的IO量；而如果业务层决定应用层需要很高的异步IO发送量，那么此时应用程序为何不一股脑地将IO下发下去呢？积压到硬盘处可以提高IOPS，但是如果积压在应用程序处，不但提高不了IOPS，反而IO延迟一样也还会很高，积压在哪里不是积压？最终还是要看应用层所感知到的IO延迟。

综上所述，为了保证整体较高的IOPS，底层的Queue最好处于积压状态。如果不让IO等待，不排队，那么整体系统也就无法达到高IOPS，此时虽然单个IO的延迟是最低的，但那是通过牺牲整体IOPS所换来的，是通过在应用层积压IO而不是磁盘层积压而换来的。IO迟早要被发送，早发送，等待长一些以获得较高IOPS，还是现用现发送，以获得单个IO的低延迟但是整体较低的IOPS，这个就要看实际情况到底需要哪种策略了。

羊毛出在羊身上，有时候可能会人为地降低Queue Depth以限制QL以保证每个IO的延迟不要太高。这仿佛是一个博弈过程，即存储层与应用层的博弈，存储层通过限制QD来获取比较好看的IO性能指标，但是其本质其实是通过牺牲应用层IOPS来换取的，也就是让应用降低IO下

发速度。实际上，现实中多种应用系统，其产生的IO属性大部分为同步IO，一方面是受制于业务层需求，另一方面也受制于异步IO开发时候的难度以及稳定性，除非对存储性能有人为的压榨目的考虑，开发人员不太愿意使用异步IO模式。这样，底层的Queue被充满到一定程度的几率就会比较低，那么整体IOPS也不会很高。而这种情况是很多用户所迷惑的，当初购买时使用测试工具测试时表现出来很高的性能，但是为何实际使用中却性能低下？所以，看完了上面的文字，你或许应该知道性能分析应当首先从哪里入手了。

那么是不是QD设置的越高，IOPS就会线性提升呢？显然不会，没有这么便宜的事情。QL越长，NCQ、TCQ等算法获得的效率提升比率就越大，但是这些算法是有优化极限的，而且上层下发的每批IO的优化程度也是不一样的，另外再加上芯片处理能力、接口带宽等潜在瓶颈点的限制，所以当单盘IOPS达到峰值以后，再增加Queue Length就没有效果了，反而此时延迟会大幅增加。每个设备都会提供一个最适合的Queue Depth值给它的上下游。

大部分HBA卡处的Queue一般被限制为16，以在IOPS与IO延迟之间保持平衡。有些厂商可以让用户调节QD，有些则不可以。设置QD有一定的讲究，IO上游模块的靠下游位置的QD要等于其可并发连接的下游模块的靠上游位置的QD之和，如果某IO模块下游只连接了一个模块，那么这两个模块耦合处的QD应相等。不直接耦合的两个IO模块之间的QD没有关联。系统路径上的同一个IO模块上下游位置上的QD要保持相同，但是有一点例外，比如，如果这个模块会将IO进行合并、拆分等处理的话，那么上下游QD要按照IO处理之后产生的IO数量翻倍或者缩倍的倍率来进行匹配。

第3章中曾经提到过，对于一个具有强消化能力的存储系统，一次

多个IO与一次一个IO相比可能一开始表现出的IO延迟相差不多，怎么这里两个IO时候就已经与一个IO时候相差较大了？因为这是单块磁盘，不是大存储系统，消化能力当然有区别，大型存储系统的巨量RAM缓存对单块磁盘的劣势有较好的屏蔽和优化作用。

IO Size与IO延迟的关系： IO Size越大，从盘片读写以及传输每个IO所需要的时间就越长。所以，Write Merge操作其实是会增加每个IO的延迟的，但是它提高了效率，提高了系统吞吐量。一些网络传输技术，比如Infiniband之类，其物理链路编码速率可能与其他的传输技术相同，但是延迟却比其他网络低。这里的意思就是说Infiniband的传输单元小，对于一些小的IO操作能够实现更高的响应速度，适合于高实时性的传输，但是其整体吞吐量就不一定比其他传输技术高了。

9. SATA协议中的NCQ

关于NCQ的首次介绍出现在本书的第3章中，再结合之前刚刚介绍的Linux下的IO Scheduler原理，想必大家已经对队列（不仅仅是NCQ）中的电梯和其他优化算法有了一个大概了解了。在这里再对NCQ深入一步。经过刚才的介绍大家已经了解，磁盘可以将队列中的IO进行优化重排然后依次执行，但是执行完毕每个IO之后，并不一定要立即返回这个IO的结果。按照常理应该是立即返回，但是在Queue开启之后，磁盘往往要积攒一定量的执行结果才向主机端磁盘控制器（FC卡、SAS卡、SATA控制器等）返回，这样做是为了降低频繁中断对控制器的影响以及对链路带宽的浪费。磁盘可以将多个IO执行的结果批量返回给控制器，每一批传输只需要一个中断即可，这个过程称为中断聚合。

再仔细考虑一下，主机端控制器将IO一个一个地充入磁盘的Queue

中（每个需要被Queue化的命令都会被控制器设置一个Tag值来表征这个命令，SATA队列深度最大为32，所以Tag的值可以是0~31），但是磁盘却可以将这些IO打乱顺序来执行，而且也不按照IO发送过来的顺序将结果（ACK或者数据）返回给控制器，如果没有一种特殊的机制来处理的话，主机控制器将不会分辨出磁盘返回的结果对应的已经完成的IO到底是哪些。解决这个问题必须由磁盘主动，因为只有磁盘知道实际的IO执行顺序。在NCQ中，这种机制称为First Party DMA，简称FPDMA，之所以称为“First Party”是因为DMA地址等是由磁盘来主导的。磁盘在每执行一个IO之前，会发送一个FIS（Frame Information Structure）给主机端磁盘控制器，FIS中包含了当前所执行的IO的Tag值。主机端控制器收到对应的FIS之后，便知道磁盘当前在执行的IO是哪一个了（因为是在控制器之前将IO打上Tag并充入到磁盘Queue中的），那么控制器此时就可以将这个IO对应的数据从内存对应的地址传输到磁盘（写过程），或者将数据传输给控制器从而控制器将这块返回的数据放置到对应的内存地址（读过程）。

另外，FPDMA还可以允许磁盘先传送某个IO的一部分数据，然后之间穿插传输一部分另外某IO的数据，再将之前的IO剩余数据传完，为何会发生这种场景？是因为旋转等待时间，比如有两个IO，IO1和IO2，当执行IO1时，磁头寻道至IO1所在的磁道时，刚好落入IO2所在的领地，那么此时磁盘不会浪费这个时隙，先捎带着读出或者写入IO2的内容，然后等待盘片旋转至IO1处时，再读出或者写入IO1的内容。而之前IO2的内容可能没有读或者写全（比如刚才寻道时磁头并没有落到IO2的起点地址对应扇区的起始字节），那么之后磁头继续去读IO2，此时如果有IO3可以获得优先“捎带”式的优化，就重复刚才的过程。如此复杂的机制，全部都是为了底层时隙被充分利用。

10. 面向SSD的Queue优化

对于NAND Flash颗粒，单个逻辑颗粒（一块物理颗粒中可以有多个逻辑独立的部分，暂称逻辑颗粒吧）所能达到的读写速度或者IO是很低的，比如20MB/s到30MB/s的样子。那么为什么SSD可以达到将近280MB/s的读速度以及30000~50000左右的IOPS呢？除了SSD内部有较大的RAM缓存之外，还有一个根本原因就是SSD控制器后端挂接的Flash颗粒都是并行执行IO的，而不是像机械硬盘那样由一个磁头来串行执行IO。如图19-74所示为一个SSD的典型内部逻辑框图。8片Flash芯片颗粒，每两个挂在一个通道上，每个通道为40MHz速率的8bit位宽的通道，也就是说每个通道的速率为40MB/s。4个通道共同连接到Flash控制器ASIC芯片上。

19-74 SSD内部典型框图

每个Flash芯片其实还可以划分为更小的单元，也即是Die和Plane，每个Die可以包含多个Plane，一般为两个。每个Plane都有各自的Page寄存器用来存储读出或者待写入的Page数据，所以每个Plane可以同时服务于一个IO。如果每个Flash颗粒有两个Die，那么每片就可以同时服务于4个IO，也就是并发4个IO，那么8片Flash就可以并发32个IO，也就是每个通道可以并行各下发4个IO到其后端下挂的每片Flash芯片来等待执行。这里可能会产生疑问，后端通道只有8b宽，这么窄的通道，每个IO一定是串行在通道中传输的，何谈“并行”？这个问题与FCAL Loop中如何实现并发IO的本质是一样的。可以参考本书附录中的Q&A相关部分得到答案，即目标存储介质执行IO是需要一定时间的，对于机械硬盘，这个时间间隙（时隙）表现为寻道+旋转延迟。而控制器利用这个时隙，将多个IO下发到多个目标上由多个目标并行执行（各自寻道，这个

可以微观并行），这就是并发，虽然在传输时一定是串行的，但是执行时却是并行的。那么对于Flash芯片，由于没有寻道操作，难道它也会产生时隙让控制器“有机可乘”，然后分派多个IO下去么？一样有，Flash芯片执行IO时有一个潜伏期，不管是读还是写，写需要的潜伏期比读更大。就是利用这个潜伏期，使得控制器有充足的时间可以通过串行通道串行地下发多个IO给后挂的每个Flash芯片，每个Flash芯片并行地执行IO，然后串行地将结果返回给控制器。

图19-74中所示的CE，即Chip Enable，中文叫“片选”，是用来让控制器选择IO目标的。每个Die对应一个片选地址，控制器通过片选来定位到每个Die从而下发IO（可针对一个Die中的多个Plane并行下发IO）。

随着集成电路技术不断发展，SSD的规格越来越高，片选的密度越来越高。目前最高的密度是每个Flash芯片包含4个CE，每片2.5英寸的SSD，其电路板正反面每面焊接8片Flash，共16片Flash就可以有64个片选，每个片选可以定址一个Die，每个Die如果有两个Plane，那么就可以并发128个IO。这一点是机械硬盘想都不敢想的（机械硬盘虽然有多个磁头，但是同一时刻只能有一个磁头在读写，只能并发一个IO，或者说不能并发IO）。将来可能出现能并发更多IO的更高规格。

能并发多少IO，就意味着其前端的Queue Depth至少要多深。并发128个IO，那么QD至少要有128才能满足胃口。而SATA协议标准中，NCQ最大的QD不过32而已，已经无法满足SSD日益增加的胃口。对于此，不少SSD正转为使用SAS接口，因为SCSI协议标准中的TCQ最大深度为256。不仅如此，主机SAS、SATA控制器也要跟上这一变化，需要从Firmware和Driver两个层面进行Queue优化操作。

如图19-75所示为刚才那款SSD的测试结果，100%的4KB随机读，thread为线程数，每个线程产生一个同步IO，所以整体产生的并发IO请求等于线程数，也等价于队列深度。可以看到1个IO的ART为0.2ms，一直到16个并发IO的ART为0.6ms，翻了3倍。而对比机械硬盘的测试结果，有天壤之别，其原因就是因为SSD内部是并发IO的。但是从16~32之间却有了质的变化，可以看到32时的ART相比16时翻了一番，IOPS增加的比例却微乎其微，此时表明这块SSD的IOPS已经达到饱和，那么可以判断出，这个SSD的最大并发IO数为16，也就是有16个片选。此时再往队列中充入IO，那么多于16个之后的IO只能等待，所以延迟翻倍。从测试结果还可以看出，SSD就算再差劲，其平均延迟是远远低于机械硬盘的，而IOPS则是远远高于机械硬盘的。

图19-75 SSD队列深度与IOPS曲线

另外，SSD的并发IO与机械盘组成的RAID一样，也是有一定几率的，在地址完全平均分布的随机小块IO的场景下，盲并发几率达到最高。SSD一样会对充入Queue中的批量IO进行优化算法，控制器会分析Queue中的IO，看看其中哪个IO目标地址正好落在了一个空闲的Flash颗粒片选上，则立即将其下发。其首要原则就是不让任何片选空闲，所以，SSD执行IO的顺序也是乱序的，因此需要NCQ的FPDMA来支持。

至此，应用程序发起的IO请求在内存中经历了文件系统、卷管理系统、设备驱动的层层历练之后，终于出了内存，被放到了IO总线上进行传输。出了内存之后，IO请求首先被送到了PCI控制器然后立即送上了PCI（PCIE、PCIX）高速总线快车道，下一站目的则是位于总线那端的HBA适配卡。进入适配卡的内存后，经由适配卡上处理器的调度，立即又被发送到了适配卡后端所连接的浩瀚网络中。至此，这个IO请求终于

跑出了被禁锢已久的主机，游弋于浩瀚的网络海洋之中。殊不知，等待它的，将是另一个洞天！

19.2 理解并记忆存储端IO路径架构图

本节论述IO的目的地，也就是存储系统端的IO路径模块架构。IO在起源地主机端经历了一系列复杂的操作之后，最终被发送给存储端处理。任何上层的以及任何类型的IO，在离开主机端之后，都会以Block IO的形式发送给存储系统处理（基于Block协议的存储系统）或者以网络文件IO的形式发送给NAS存储系统处理。IO请求可谓是方出龙潭又入虎穴。“存储端”这三个字有三种理解程度：“本地磁盘”、“主机端本地RAID”和“外部磁盘阵列”，其中前者与后两者有着本质区别，后两者之间并没有本质区别。对于前者，从设备控制器驱动程序下发的IO请求会直接被控制器透传到目标设备；而对于后两者，驱动程序下发的IO请求被控制器接收到之后，控制器本身还需要做一定的虚拟化映射处理，最终发给目标磁盘设备的IO请求在内容上和数量上都不一定与驱动下发的IO相同。

外部存储系统对IO的处理过程与主机端有很大的同源性。可以说存储系统本身就是一台主机，软件架构上与主机相同，都是操作系统+应用程序组成的，存储系统的操作系统，有些基于Linux，有些基于Windows，有些基于UNIX，有些则基于VxWorks等。

下面分别描述一下存储系统中的各个层次。

19.2.1 物理磁盘层

主机端发送的任何数据最后都要存储于物理磁盘中。

对于一个高可靠性存储系统来说，磁盘的Cache必须被设置为Write Through模式，因为物理磁盘没有任何电池保护措施，一旦失去电源，

磁盘Cache中的尚未写入盘片的数据将会永久丢失；而如果Cache运行在Write Back模式下，就将造成数据不一致情况的发生。对于不支持永久Write Through模式的磁盘Cache，只能靠磁盘控制器或者驱动程序来强行设置每个IO的FUA位来通知磁盘进行Write Through模式的写入。

而对于磁盘自身的读Cache，控制器不会干预，而且所有存储系统都会允许磁盘自身的Cache对任何读操作做缓存，甚至还可以进行磁盘自身的预读。如本章前文所述，Write Through之后的写数据会一直留在磁盘自身Cache中，作为读缓存数据而存在，这里需要区别一个细节，即缓存中的预读数据和被动式的读数据，刚才说的这种Write Through之后被动留存在缓存内的数据称为被动式读数据缓存，而磁盘缓存自身从盘片内预读入的数据为主动式的预读数据。

磁盘读缓存对于小块连续IO的效果是非常显著的，如果在主机端发起512B的连续地址IO，则在存储系统处即可以监测到单块磁盘本身接受和处理的IOPS甚至可以达到高于1000，这显然是产生了Read Cache Hit导致。在综合属性的IO下，单块磁盘的IOPS一般在200~400之间。

支持NCQ或者TCQ的磁盘本身具有Queue机制，控制器可以将一批IO充入磁盘Queue中从而让磁盘自身来对Queue中的IO进行优化操作。

盘片大小与性能的关系

目前，业界新推出的存储系统都已经推广使用2.5寸磁盘，3.5寸磁盘由于其耗电量太大，占用空间大，已经不能满足当今时代节能降耗的需求了。目前2.5寸的SAS磁盘其容量甚至比3.5寸SAS盘还要大，目前最大到900GB。并非技术上3.5寸的不能做到更大容量，而是厂家故意使然，即推广2.5寸磁盘。那么盘片变小了，其性能是否会减弱？答案是

肯定的。多种原因，首先2.5寸磁盘其目前最大转速为15krpm，但是相比10krpm的2.5寸盘来讲，成本大幅提升，所以目前投入使用的都是10krpm的规格。而3.5寸盘基本上都使用15krpm规格的，转速上低了一筹；其次，小盘片其线速度会比大盘片低，加上转速低，两两加成，导致性能下降。但是我们知道，盘片越小，磁头摆动周期就会越短（并不是说磁头摆动的速度高了，而是需要摆动的距离短了），同样的容量密度下，2.5寸盘片比3.5寸盘片在平均寻道时间上要低，而寻道时间恰恰是影响IO性能的重要指标，那么是不是2.5寸的比3.5寸的IOPS性能要高呢？但是还需要考虑一点，现在的磁盘对队列的支持都非常成熟了，虽然论单个IO的平均寻道时间，2.5确实比3.5强，但是一旦外部IO被队列化之后，那么单IO平均寻道时间这个参数就会被强烈的屏蔽和弱化。因为在队列机制的介入下，磁头一般都是顺着一个方向摆动就可以捎带执行多个IO，而不是来回频繁寻道摆动，既然这样，不管盘片多大，哪怕10寸的，其向一个方向持续摆动时执行IO的效率，就取决于盘片旋转时的线速度了，线速度越快，旋转延迟时间越少，IOPS就越高。所以，在性能上，2.5寸盘输给了3.5寸盘，但是在能耗上，前者比后者可以降低40%左右。成本上，目前2.5寸盘依然高于3.5寸盘，相信随着时间的推移，2.5寸盘的成本会逐渐下降。

19.2.2 物理磁盘组织层

多块磁盘被插入到一个扩展柜中进行集中供电散热和监测。扩展柜都会为每个磁盘提供两个接口并支持任何时刻从任何接口接收或者发送数据。扩展柜上的控制模块的作用请见本书前面的章节介绍。

扩展柜控制模块本身可以说是一个嵌入式系统，其有自己的CPU、内存、Flash ROM永久存储芯片，软件上，它有自己的操作系统（一般

以Firmware的形式存放在Flash Rom中)。扩展柜控制模块掌管着其上磁盘的数据传输以及监测，所以控制模块硬件及Firmware是否稳定，关系到整个存储系统的可用性，有些控制模块被设计得很山寨，整天不是这坏就是那坏，硬件坏完了就出Firmware Bug，没完没了的问题。控制模块一旦发生问题，轻则监测不到硬盘但是硬盘数据IO正常，重则整个扩展柜的磁盘直接与机头断开，无法访问到。

通常情况下，扩展柜磁盘的数据IO与监测IO是分开的两条路径，控制模块上有一类关键芯片，也就是FC Loop Switch芯片，低端一些的只用PBC (Port Bypass Circuit) 芯片。前者在物理上是交换直通架构，后者则是手拉手环Loop架构。每个磁盘都与这个芯片连接，芯片的上行通路或者直接与机头上的适配卡连接或者连接到上行级联扩展柜的同样芯片中，数据IO的直接控制者是机头上的适配卡和它的驱动程序，Loop Switch芯片只起一个物理数据传输的作用以及FC Loop仲裁响应等作用。所以只要这个芯片正常运作，控制模块上其他地方出现问题的话，那么至少可以保证数据IO是正常的，至于一些监测动作，比如磁盘温度的探寻、各种传感器数据探寻以及与SES有关的探寻等可能都会变失效。但是一旦不稳定的Firmware发生崩溃导致整个控制模块重启，那么此时会连数据IO都不可进行，机头上的操作系统会认为这种情况为磁盘丢失，受到影响的RAID Group全部需要被Rebuild，这无疑是一个巨大的浪费，不但影响了数据访问性能而且还降低了磁盘寿命，更严重的情况还会导致连环灾难，即在Rebuild过程中再次发生丢盘，一旦一个RAID 5组中丢多余2块盘，那么此时整个RAID组的数据就会宣告丢失了。

总之，扩展柜控制模块的设计一定要力求稳定。其次在连接线缆方面也需要注意，有些时候因为线缆问题而导致整个柜子无法访问。磁盘

阵列后端连接所使用的线缆多种多样，有各种各样的铜缆和光缆，虽然它们都承载着FCAL协议。

19.2.3 后端磁盘控制器/适配器层

磁盘控制器以一个适配卡的形态存在于存储系统机头当中，当然有些适配器是集成在主板之上的，但它依然是通过PCIX或者PCIE总线与系统桥芯片连接的。适配器的每个接口都会连接一个扩展柜上对应的接口，由于扩展柜上提供了双Loop接口冗余，所以一个机头上可以同时使用两个适配器接口来连接同一串扩展柜，这样既可以实现路径冗余又可以实现链路带宽均衡。当然，不是所有存储系统都支持这样做的，扩展柜上的双接口往往是各自连接不同的机头，两个机头之间做冗余或者负载均衡。

机头上的操作系统执行与主机系统类似的过程，通过驱动程序，从所有适配器上发现后端FCAL等网络上的Target设备，每个Target就是一个物理磁盘。

注意：任何存储系统绝对不会使用磁盘自身的Write Back模式的缓存，因为一旦掉电，数据不保。所以后端磁盘控制器要在初始化时确认所有磁盘的缓存模式，如果是Native Write Through模式，那么控制器可以不做任何担忧，但是如果是Write Back模式，那么控制器必须在每个写IO指令中将FUA位置位以强迫磁盘进行Write Through。但是一般来讲，稍微高端一些的存储系统所使用的磁盘Firmware均为硬盘厂商定制的，这些Firmware会考虑到这些的，一般都会支持Native Write Through模式。但是对于一些低端存储系统，所使用的硬盘可以是市面上的桌面硬盘，它们一般都是Write Back模式的写缓

存，对于这些存储系统的控制器就需要注意了。

19.2.4 RAID管理层

机头上的操作系统识别到了所有后端的磁盘，下一步它需要对所有这些磁盘来划分RAID Group。可以说划分RAID组完全是一个软件行为，与任何硬件无关。通常所说的所谓“硬RAID”都是极具误导性的说法。所谓硬或者软，是指对客户端主机系统来讲的。任何不耗费主机CPU处理资源所实现的功能，对于主机来说都是硬，比如硬解压卡、硬RAID卡等。至于外部智能磁盘阵列，其对RAID的管理当然也不耗费主机端CPU资源，所以也可以称为一个超大硬RAID卡了。

但是对于存储系统本身来讲，硬RAID这个词没有任何意义。操作系统可以使用各种方式来记录RAID信息，比如直接记录在所有磁盘上的一段固定保留区域，或者记录在存储系统控制器的Flash ROM中，但是现代存储系统都会选择同时保存在磁盘固定区域中和Flash ROM或者任何其他形式的永久存储介质中各一份。这样做的好处就是当把磁盘拔出再插入时，系统可以根据磁盘上所记录的信息来判断这个磁盘为尚未加入任何RAID组的磁盘，还是已经存在的某个RAID组中的磁盘，而不管这个磁盘插在哪个适配卡接口下的哪个扩展柜。

Flash中保存的那份则用来在系统启动时与磁盘上的信息作比对看看是否一致，如果不一致则会用某种方式提示用户来选择使用哪一份。这种功能的一个应用例子就是，比如两台同样型号的存储系统，某时刻管理员想把系统A上的一整个RAID组迁移到系统B上，那么管理员需要把所有系统A上对应这个RAID组的所有磁盘拔下来然后插入系统B上，系统B的操作系统此时就会读出这些新插入磁盘上所保存的信息来与当前Flash中保存的信息做比对，发现Flash中并没有这些记录，所以系统B就

会认为这是一个新的RAID组，经过一些配置之后，系统将Flash中的信息与当前信息进行同步。

系统一方面可以让用户手动选择用哪些磁盘组成一个RAID组以及RAID的类型，另一方面也可以自动创建RAID组，用户只要给出这个组中需要包含多少磁盘，系统就会根据最优条件自动创建。比如尽量将组中的磁盘分布在不同的后端适配器接口的不同扩展柜中，尽量平衡。

关于RAID条带化编址。条带化其实有两个含义，第一是逻辑地分割每个磁盘为多个Segment，每个磁盘相同位置的Segment组成一个横向条带，然后计算校验的时候，以条带为单位。条带的另一个含义，也是最重要的含义，是条带把本来的纵线编址映射为了横向编址，即整个RAID组逻辑空间对应的物理地址是以条带为单位向下排列的，条带内则是以Segment为顺序自左向右的，而Segment内则又是自上而下的顺序。这种纵横结合编址的目的就是为了让某个IO请求的地址段可以跨越整个条带，这样，每个IO就会有多个磁盘为它服务，提高速度。这种设计思想对于某些情况确实可以提高性能，但是在另外一些条件下，性能不升反降，所以，这种思想目前正在被逐渐打破，各种新模式正在逐渐出现。

关于存储系统使用的XOR芯片。XOR芯片只是作为一种外设与存储系统机头硬件的IO总线所连接，比如完全可以使用PCIE总线来连接这个XOR芯片，任何需要进行XOR的运算操作，系统都会将数据发送至这个XOR芯片来计算，计算完成后再放入系统内存。由于XOR芯片为ASIC设计而不是普通CPU设计，运算方式有本质不同，所以大大减轻了存储系统主CPU的压力，所以可以说XOR芯片为存储系统的一种硬加速器了。

RAID组划分本身是一个非常迅速的过程，因为它并不涉及很多实际数据的IO操作，只是将分组信息数据写入磁盘固定位置即可。最耗费时间的是划分之后的RAID组初始化过程。关于RAID初始化的细节请参考本书前面的章节。

最后，创建好的RAID组可能还需要为其指定一个首选控制器，即选择由哪个机头控制器来负责对组内所有磁盘的数据IO操作。由于高可用的存储系统一般都具有双控制器机头，每个机头都连接到同一串或者几串扩展柜，为了避免发生数据一致性问题，同一个磁盘任何时刻一般只允许由一个机头访问，所以机头一般会向对应的磁盘发起SCSI Reservation操作。创建RAID组之后，所有组中的磁盘只能同时被一个机头来Reserve，当一个机头发生故障之后，另一个机头会强行接管所有的RAID组。这种机制的存储控制器一般都是松耦合的，比如NetApp的FAS系列存储。但是目前几乎所有双控存储系统都允许双控同时访问同一块磁盘了，而只是将其上的LUN来分配给单独一个控制器机头管理，另一个作为冗余。同一个磁盘上可能会存在多个LUN的数据部分，而这多个LUN可能又隶属于不同的控制器管理，所以此时就需要双控可以同时访问同一块磁盘。

存储系统创建RAID组的目的有3个：第一个是为了防止物理磁盘损坏导致的数据丢失；第二个是为了提高数据访问速度，做对应的条带化，提高并发度；第三个是将底层的大量磁盘再分割成较小的逻辑空间，也就是RAID组，方便管理。

1. RAID的Write Hole现象及应对方法

如图19-76所示为Write Hole现象的示意图。某时刻某条带上正有一个Segment（D1）以及相应的Parity Segment（P）需要更新，但是在更

新过程中，突然断电。此时磁盘上的数据状态可以是以下6种中的一种：D1'以及P'均被成功地更新到了磁盘上、只有P'被更新、只有D1'被更新，D1'只写了一部分、P'只写了一部分、D1'以及P'均只写了一部分。具体是哪一种，系统不得而知。至于“只写了一部分”是什么意思呢？大家知道磁头是要在磁盘上进行磁化操作以写入数据的，所谓部分写入就是指磁头只磁化了对应Segment中的一部分扇区，甚至断电前磁头所位于的扇区本身也只被磁化了一小段或者一大段，并没有全部磁化，这就是所谓的“部分写入”。

图19-76 Write Hole现象示意图

上述6种状态中，只有一种是需要的理想状态，其他都为不一致状态，比率为1:5，所以系统突然断电之后，数据便会有很大几率处于不一致状态，系统重启之后必须对这种情况进行处理。断电之前未写入成功的数据，磁盘不会返回成功消息，那么控制器就会感知到断电之前的状态。在电力恢复之后，控制器会将上次断电前未写入成功的数据再次覆盖写到相同的地址，以确保写入成功。但这样做的前提是控制器中的数据缓存具有电池保护，或者可以在断电之前将缓存中的数据转存到永久性存储介质比如Flash卡或者磁盘中存放。同时，系统需要使用某种状态机来记录每次更新条带的结果，更新开始时将条带标记为“no_sync”，表示本次更新尚未完成，如果此时掉电，那么重启之后系统可以感知到这个条带处于no_sync状态，则对其进行重新写入，写入之后将其标记为in_sync，即条带此时是一致的。每个条带的状态（no_sync/in_sync）都被记录到单独的元数据链中保存。

2. RAID层可以考虑的一些优化设计

有时候，一个RAID组的磁盘上可能只承载了一个或者几个LUN，

而如果一个RAID组中的所有的LUN所占据的空间与RAID组全部空间相差很大的话，那么当RAID组中某盘故障后插入新盘开始重构的时候，就没有必要将全部条带都进行重构，完全可以只重构那些被LUN所占据的条带。当然，前提是这些LUN在RAID组成员盘上的位置都是固定的，而不是随时变化的。这样就需要RAID管理层与LUN管理层相互配合来实现这个目的，这样就会大大降低重构时间，延长设备寿命。目前已经有一些产品实现了这个功能。

RAID层在管理RAID 0时可以采用更加灵活的方式。如图19-77左侧所示，有三块容量各不相同的磁盘，如果用它们来组成RAID 3/4/5/6这种校验型的RAID，那么系统必定会以容量最小的盘为准，其他盘容量比这大的部分将会被截断而不用。但是对于RAID 0，无须校验，所以也就不受这个限制。图19-77右侧所示的RAID 0布局方式中，有三大块区域，被称为Zone。

图19-77 灵活布局的RAID 0系统

此外，RAID管理层还负责扇区重定向等操作，详见本书之前章节。

19.2.5 LUN管理层

RAID组划分之后，操作系统对待每个RAID组就是一个逻辑上的连续地址的存储空间，如果用操作系统驱动链的眼光来看，那么每个RAID组也都会有一个驱动程序，这个驱动程序负责将RAID组逻辑空间地址翻译映射为底层物理磁盘的物理地址，然后将对应的IO请求传递给磁盘控制器驱动程序。

在RAID组之上，存储系统还会生成另一层逻辑层次，即LUN，

LUN也是存储系统最终虚拟出来的最后一层东西。LUN会被直接映射给客户端主机使用，客户端主机在其自身存储适配器总线扫描的过程中会发现一个Target中的若干LUN。

前文中提到过，存储系统会将每个RAID组视作一个连续地址的逻辑存储空间，也就相当于一个大逻辑磁盘。那么LUN在这个逻辑空间之内到底是怎么分布的呢？是像普通操作系统分区一样，按照连续的地址一段一段地来分割，还是可以像文件系统一样，每个块可以分布在逻辑空间内的任何区域内，然后用一个复杂的数据结构来将这些分散的块组合起来形成逻辑上连续的LUN呢？这个问题并没有严格的答案，每个厂商的实现方法都不同。有些厂商就是使用前者的方式，即每个LUN在RAID组空间内都是连续的一大块；而有些厂商为了获得极高的灵活性，直接采用了某个文件系统来管理LUN，也就是说，LUN在这个文件系统下面就是一个文件，这个文件可以存放在底层逻辑空间内的任何区块中，可以连续，当然也可以随机分布。使用文件系统来管理LUN的一个典型例子就是NetApp的WAFL文件系统，这里就不做过多描述了。所以说，存储系统内部也有卷管理系统或者文件系统，这两个模块将底层逻辑空间再映射分割为上层的逻辑空间，也就是LUN。

下面还是用几张示意图来描述LUN在RAID组内的分布方式吧。总结了一下共有4种分布方式。虽然在具体设计实现的时候又会有细节的不同，但是总体的思想就是这4种。

1. 纵向非条带化LUN分布方式

这种模式下，每个LUN的逻辑地址是纵向分布的，即用完了第一块磁盘再用第二块磁盘，但是底层的RAID组还是以Stripe方式来做逻辑分割并且计算校验值的，所以这种方式下的LUN与条带是垂直的。这种

LUN分布方式在一定条件下是非常合适的，具体见下文。如图19-78左半部分即为这种方式的示意图。

图19-78 条带和非条带LUN分布方式

2. 横向条带化LUN分布方式

这种LUN分布方式是大多数产品都在使用的设计。LUN与Stripe平行。这种分布方式在一定条件下会显示出非常低的性能，具体见下文。如图19-78中右半部分即为这种方式的示意图。注意，图中的Parity磁盘只是示意图，对于RAID 5来讲Parity是分布式的。

另外值得一提的是，有些产品设计要求LUN在磁盘上的分布必须连续，不可打断。比如上图中的例子，如果删除了某个LUN，那么对应的这块空间将会被标记为空闲，而如果随后需要创建一个LUN，但是这个LUN却比空余的这块空间要大，那么就不能够直接使用这块空间，而必须在整个空间尾部划分一块新的空间给这个LUN；如果整个空间内已经没有连续的空余空间来容纳这个LUN，但是零散的空余空间加起来却可以容纳这个LUN，那么此时系统会做类似文件碎片整理的动作，将某个或者全部的LUN前移或者后移，将缝隙合并，空出连续的空间。这样做的根本原因就是因为这些产品所设计的LUN映射管理系统并没有足够的智能来在不连续的底层空间来做映射。而有些产品则可以做到。

3. 类文件系统LUN分布方式

这种分布方式与文件系统无异，只不过每个文件可能比较大，比如几十GB，几百GB，甚至上TB。LUN其实本来就是一个Block集合，逻

辑上连续，物理上可以不连续。这种LUN分布方式其实就是Filedisk思想，再比如经常用的光盘文件镜像ISO格式。如图19-79所示即为这种方式的示意图。

图19-79 类文件系统LUN分布方式

当然，这张图所表示的分布情况并不一定就是实际情况，而是一个极端情况。可以看到每个LUN的碎片都很多，而且都是随机分布的，这种情况可不是好现象，过于随机的分布会导致当执行连续地址IO时性能大打折扣，而连续IO本应该是性能最好的IO方式。

4. 基于分布式RAID的LUN分布方式

前文介绍了存储系统划分RAID组的3个目的。但是我们仔细从宏观的角度想一想，如果不划分RAID组，又会是一个什么样子呢？会不会有本质区别呢？也就是说，如果将系统中的所有物理磁盘组成一个大的逻辑空间，然后直接让LUN分布到其上，是不是也可以呢？

如图19-80所示即为这种分布式RAID的示意图。所谓“分布式”RAID，指的是一个RAID组无须独占某几块硬盘，可以与其他不同类型的RAID共享同一批物理硬盘，另外，同一块磁盘可以承载任何一种RAID类型的一部分。

图19-80 分布式RAID模式的LUN分布

逻辑空间映射管理的最高修炼成果就是文件系统，同样，存储系统中LUN映射管理的最高修炼境界也就是文件系统。我们来举例说明一些厂商修炼的结果。

入门代表：多数常规设计，即图19-78所示的两种LUN分布和管理方式。

进阶代表：WAFL。WAFL是绝对细腻的纯文件系统，甚至实现了File In File的设计，但是其底层仍未脱离RAID Group。炉火纯青代表：XIV、VRAID。XIV已经完全实现了灵魂脱壳而游刃有余，它并没有RAID组的概念。部分原因是因为它只能实现镜像式RAID 10，只不过是分布式RAID 10，没有了RAID组的概念。在物理磁盘出错时，它可以只恢复被实际数据所占用的空间而不需要全盘恢复，也就是说上层逻辑空间的分布完全游离于所有物理磁盘可以任意映射而且也不依附于磁盘，关于XIV的具体细节请参考本书前面的章节。HP EVA系列存储系统中用的VRAID的设计模式与图19-80所示类似，但是它尚未做到XIV那样的只恢复被占用的数据。

提示：目前市场上的产品大多数还是使用横向条带LUN分布方式。纵向的分布方式几乎没有人如此设计。分布式RAID的LUN分布方式也没有产品能做到如图所示的那种灵活程度，HP公司VRAID只是一个初级版。但是类文件系统的LUN分布方式，WAFL、XIV、3PAR等都已经可以做到，而且WAFL也是有潜力做到以任何方式分布LUN并且提供用户配置接口的潜力的。完全块级虚拟化的LUN分布方式是将来的趋势。

5. 全打散式LUN分布方式

上述的几种LUN分布方式都是局限在某个RAID组中的。比如系统中共有100块盘，每10块做成一个RAID 5组，然后在各个RAID组中再划分LUN，每个RAID组之间的LUN互不影响。划分成多个RAID组有利于性能和容量隔离，组与组之间在容量与性能上不会产生冲突（忽略底层

传输链路以及系统总线等底层冲突)。然而,有另一些新型的开放式存储,比如IBM的XIV以及3PAR等,它们的LUN是被打散成小块(比如XIV粒度是1MB)然后将这些块均衡地分散在系统中的所有磁盘中。这种分布方式的优点是巨大的,在应对随机IO方面提速很多。理论上讲,如果将LUN均衡分散在所有磁盘中,系统中如果有180块磁盘,那么系统理想状态下最大可以并发针对这个LUN的180个随机IO,实际中并发几率也很高(并发几率随着盘数提高而提高),每个盘同一时刻都服务于一个IO,而如果只将这个LUN分布到比如10块盘组成的RAID 5组中,理论上最大只可以并发5个随机IO,而且并发几率很低。线性提升随机IO的性能是这种全打散的LUN分布方式的最大好处。

但是这种分布方式也有一个致命的缺点,即在面对大块连续IO的情况下,系统很难受。如图19-81所示为某采用全打散式LUN分布设计的产品。假设某LUN的物理块顺序为1234567,这7个块的分布如图所示,系统中会为每个块保留一份冗余块。当某应用发起读取从1~7这7个块的时候,本应为连续IO,但是此时系统所做的事情就完全不是连续IO的行为了。由于系统并不是按照严格横向条带化方式分布LUN,所以原本连续的块可能随机地分布到每块硬盘的不同位置,这样,原本连续IO就变为了随机IO。虽然此时仍然可以多盘并发提供服务,但是每个盘都要随机寻道来读写它所拥有的分块。传统意义上的“连续IO不需要频繁寻道”的结论不再适用这个情况了。

图19-81 全打散式LUN分布模式

主机发起对这个LUN上的这7个数据块的连续读IO操作时:

(1) 先从一个柜子中的磁盘上将1和2数据块读入缓存(磁盘需要多次寻道)。

(2) 再从另一个柜子将3、4、7数据块读出（磁盘需要多次寻道），并且通过内部以太网矩阵传递给接收主机IO请求的那个柜子。

(3) 再从另外一个柜子中将5、6数据块读出（磁盘需要多次寻道），并且通过内部以太网矩阵传递给接收主机IO请求的那个柜子。

(4) 接收主机IO请求的柜子收集到所有数据块之后，将数据块返回给主机。

以上只是在单LUN的连续IO方面，而在面对多LUN并发的大块连续IO时，整个系统性能骤降。本质原因就是因为在LUN被均衡打散，大块连续IO必定会在单位时隙内占用几乎所有磁盘为某个LUN服务，而此时如果有第二个LUN也接受大块连续IO，那么所有的磁盘也同时需要服务于这个LUN，这样这两个LUN就产生了资源争抢，系统到底优先服务于哪个LUN呢？这就像一个十字路口，红灯到底亮几秒才合适呢？不管怎么样，系统肯定不会以太大的时隙粒度来轮流服务每个LUN，这样就必然导致雪上加霜。对一个LUN的连续IO访问已经牵动了所有盘，更别提同时服务多个LUN的IO了。

所以，全打散的LUN分布方式，最好提供给用户以可控的接口，让用户来定义具体某个LUN如何在系统中的磁盘上进行分布。针对不同的应用系统，选择不同的分布方式，而不是一竿子定死。目前据笔者所知只有中科蓝鲸的BWFS提供横向条带化与纵向顺序分布这两种文件分布方式（文件分布与LUN分布本质是相同的）。

另外，为了避免磁盘频繁寻道的问题，全打散式的LUN分布可以选择尽量将逻辑上连续的LUN块分布到尽量连续的物理磁盘块中，这样最起码可以保证单LUN大块连续IO情况下性能不打折扣。这样做就相当于

横向条带化的分布模式了，只不过是横跨到系统中的所有磁盘中了，但是这样做也同时降低了LUN分布的灵活性。但是不要误解一点，横跨系统中所有盘并不意味着所有盘组成一个大的RAID 5组，可以组成多个RAID 5组，LUN分布于底层传统RAID是分开的两个层面。应始终牢记，不管LUN如何分布，RAID只管在组内磁盘的横向相同地址上的数据计算校验并保存，RAID是防止单盘失效的，不要将其与上层的東西混淆。

6. LUN分布方式的设计对其他阵列高级功能的影响

LUN分布的方式直接决定着一个产品后期对Snapshot、RAID组扩容、Thin Provision以及动态数据分级（Dynamic Storage Tiering, DST）的开发难易度以及粒度。这个问题很显然，如果采用横向条带化定死方式来分布LUN，那么其他一切高级功能的设计也就受到很大局限了。

RAID组扩容：首先来看RAID组扩容应该怎么处理。当RAID组内新加了一块磁盘之后，为了保持所有LUN依然横跨所有盘，而且还必须保证物理地址或者逻辑地址的连续性，那么系统可以选择使用两种方式來填充新加入的空余空间。第一种是完全笨办法，就是将Segment一个一个地往上移动，从而填补空隙，这个工作是耗费大量IO和计算资源的，读出、写入频繁，写入时还需要计算XOR，但是此时的XOR计算读惩罚要相对少一些，因为根据公式新数据的校验数据=（老数据EOR 新数据）EOR 老校验数据，此时老数据=0，系统事先就知道了，不需要读盘操作，但是老校验数据还是要读出的。进行重构的同时还需要继续接受上层下发的IO操作，对应用不能有影响（具体机制可以参考本书之前章节以及附录1中对RAID初始化过程的描述）。如图19-82所示为扩容前与扩容后的分布图。虽然这个办法很笨而且耗费大量资源，

但是其可以保证LUN物理地址的连续性，一了百了。

图19-82 笨办法扩容

第二种办法则是不移动实际数据，磁盘加入之后，首先进行重构，将Parity重算成一致的（由于是在线扩容，同时可以接受主机IO，所以必须重算Parity；如果是离线扩容，那么只需要向磁盘发送Zero Disk指令让磁盘自己将自己的数据清零，之后就不需要重算Parity了，因为根据XOR算法，加入0x00之后，原有的Parity值不变），之后就放那不管了。因为加入新磁盘对之前的LUN容量是没有影响的，之前的LUN完全可以选择不动地方。如果此时用户想在这块剩余空间或者原有磁盘的剩余空间内创建新LUN，首先选择在原有空间内横向分布这个LUN，如果空间不够，那么拿这块新加入的纵向空间来补齐，这样会造成LUN的物理地址并不是按顺序横跨在物理磁盘上的，就需要做一个映射表来影射不规则的物理地址到规则的逻辑地址了。如图19-83所示为这种思想指导下的扩容前后布局图，可以看到LUN3已经被分成两部分，横向部分横跨原有磁盘，空间不够的话就支出一个分部到新加入磁盘中纵向分布。这么做的好处就是除了重算Parity外不耗费其他资源，但是会永远维护一个地址影射表，而且同一个LUN的不同部分有不同的性能。属于一种野路子。

图19-83 地址映射方式扩容

相对的，如果使用类文件系统的块级虚拟化方式来分布LUN，那么应对RAID组扩容根本不是问题，甚至可以除了重算Parity外什么都不用做，为何呢？因为这种LUN分布方式其原本就维护了比较复杂的块映射表，而且分块粒度恒定，分布的有章法有套路，虽然上面那个野路子也有点这个意思了，但是毕竟还是野路子。

同时，如果选择使用纵向非条带化模式，那么面对这种情况也根本不用做什么，除了Parity重算之外。

- **Thin Provision:** 对于横向条带化的LUN分布模式下如何实现Thin，请参考本书之前章节。类文件系统的LUN分布方式虽然可以天然地实现Thin，但是依然面临性能问题，它的性能问题与是否开启Thin Provision功能无关，而是其这种机制天然就决定了在大块连续IO下的性能折扣（见前文）。即使对全打散模式做了改进，比如前文提到的让逻辑连续的块物理上也尽量连续，那么一旦开启Thin之后，物理上是否连续就不是我们所能控制的了，多个LUN会互相挤占，这样就不连续了，此时大块连续IO性能无法得到保证。
- **Snapshot:** 我们知道Snapshot有两种方式，CoFW、WR（或称Redirect On Write, ROW）。复制或者重定向写是有一个粒度的，比如4KB，甚至128KB等。对于小块全打散式LUN分布模式，这些小块就可以作为Snapshot的复制粒度，另外，这种模式下系统本来就需要维护一份元数据，比如各种链表、位图等来记录逻辑位置与物理位置的对应关系，而Snapshot也需要类似的元数据，所以元数据方面又可以统一整合了。这极大方便了Snapshot的设计和开发。而对于横向条带化或者纵向非条带化模式，引入Snapshot之后需要引入一套额外的元数据链，增加了开发难度和设计难度。
- **DST:** 既然是块级动态数据分级，那么天然就可以与全打散的LUN分布模式以及类文件系统LUN分布模式相融合了。只需要增加基层Tier级别的定义，比如SSD Tier0，将热点块分布到SSD中同时变更地址映射表即可，另外，再加入一层热点判断和迁移策略层即可。底层丝毫无须改变。而对于其他类型的LUN分布模

式，就需要引入全新的映射链表来处理分布在不同Tier中不同位置的LUN，增加了开发难度。

所以，LUN分布方式直接决定了一款产品的前后期开发难度、功能、性能、易用性、后期新功能开发可行性等。阵列中玩的就是LUN，除了RAID之外，其他高级数据加工功能都是基于LUN的，底层架构直接决定上层实现方式和效率。

7. 关于LUN的对齐问题

所有存储系统在创建LUN的时候都会要求用户来选择这个LUN将要被何种主机操作系统使用，这样做的原因其实是因为不同操作系统使用不同的文件系统和卷管理系统，而不同的文件系统或卷管理系统又会从磁盘的不同地址来作为文件系统管理的空间的起始地址，比如，至少不可能将LBA0，也就是MBR扇区作为文件系统的空间。

而常规情况下，存储系统中所创建的LUN都是Segment的整数倍，即Segment对齐的。但是由于文件系统或者主机端卷管理程序会从某个非0的起始地址来作为自身空间的边界，并不知晓这个LUN其实是横跨在多块物理磁盘上的，这种使用非0地址为边界，导致了FS或者VM的逻辑Block也可能横跨于多个物理磁盘。那么也就是说，FS或Cache Manager每操作一个Block或Page，就需要占用两块磁盘而不是1块，产生了惩罚，如图19-84所示。为了解决这个问题，存储系统会根据用户在创建LUN时给出的操作系统类型来动态地在LUN映射关系式内将对应的起始地址偏移量变量的值向前推移对应的扇区数量，从而达到与上层的逻辑起始地址重合。上下不对齐会导致严重的性能问题，存储系统一般会判断这种情况是否发生并且通过某些输出来通知用户，比如当存储系统

图19-84 起始地址不对齐示意图

向后某个LUN下发1个IO请求时，到了底层却变成了两个，那么就很容易判断发生了Misalign情况。

为了解决这个问题，存储系统都会针对不同的操作系统来分布LUN，将LUN中对应FS起始点的LBA地址与底层卷的Block边界对齐，也就是类似于将LUN在底层RAID Group中前移一定长度的LBA地址，这段长度会根据不同操作系统而不同。

8. LUN碎片整理

请不要误会这里所说的“碎片整理”。主机端的碎片整理整理的是LUN里的文件系统块，而阵列段的LUN碎片整理指的是阵列自身在底层物理磁盘层面整理被分裂成多块的LUN，将其合并成一个物理上连续的空间。为何会出现LUN碎片？这个问题不得不拿NetApp的WAFL来开刀了。LUN在WAFL下就是一个彻头彻尾的文件，与普通文件系统中的文件别无二致。正因如此，不可避免这个LUN会随着时间的增长就可能被分布到底层物理空间的各个位置，连续IO到了底层变为了随机IO，这是谁都不想看到的。所以WAFL提供了碎片整理程序。当然，WAFL对LUN的分布是最极端也是最容易的做法。其他厂商对LUN的分布没有如此的灵活，基本上都是连续的横向分布在底层RAID组之上的。

比如EMC的Clariion CX4系列存储中，当在某个RAID组中创建了3个LUN，之后删除了第二个LUN，那么此时便会在RAID组物理空间上留下一个空隙。如果随后打算再次创建一个新LUN，如果待创建的LUN的大小小于等于这个空隙，那么系统会将这个LUN分布到这个空隙中，这是最好的状况；如果尺寸大于这个空隙，那么系统就只能选择其他的更大的空余空间来创建这个LUN，这块空隙就不能够被利用，这个空隙就被称为“碎片”，当然这种碎片肯定不如WAFL的碎片粒度大。EMC提

供了碎片整理程序，如图19-85所示为一个空隙碎片产生的过程。图19-86所示为碎片整理之后的LUN分布状况。

图19-85 LUN碎片的产生

图19-86 碎片整理完毕的状态

在Scale-Out架构的存储系统中，很多厂商就希望LUN被打碎，比如IBM的XIV、3PAR InservT系列、EMC的Symmetrix V-Max系列，它们的LUN分布思想都是打碎分布到多个节点的存储空间中，这样来讲，相当于LUN被故意设计为碎片，用多个节点并行地服务于这些碎片。这样做在随机IO访问的情况下确实会提升效能，但是在多LUN并发连续IO的情况下，效果可能适得其反，具体分析可见本章后面内容。

思考： LUN分布方式、ThinProvision、自动分级存储，仔细想来，这三者其实是可以融洽的，因为它们三者的基础都是块级别粒度的数据分布和移动。比如，某个LUN以类文件系统方式来分布，分布粒度以1GB为单位增长，利用类似文件系统元数据的方式来记录整个LUN的映射链关系，同时，也可以容易地实现数据的自动分级，以1GB为粒度，构建一个数据库来追踪记录每个1GB区域的属性、访问频率、当前所处的层级等信息，然后定时启动分级迁移策略。

所以，在设计LUN分布方式的时候，一定要结合考虑今后的增值功能规划。如果LUN的分布方式过于定死，那么就不利于今后的Thin以及自动分级的实现了。

19.2.6 前端接口设备及驱动层

存储系统前端接口用于连接主机客户端。与主机所不同的是，存储设备会有更多种类和数量的网络接口，比如FC接口、用于iSCSI协议的以太网接口、SAS接口等。主机客户端会通过前端接口识别存储系统映射的LUN并且对LUN做数据IO或者控制IO操作。

注意： 前端接口处需要注意的一个地方就是Queue Depth，在各个层次都尚未产生瓶颈的时候，Queue Depth越大，一次接受和处理的IO就越多，系统表现出来的IOPS和吞吐量就会越高。但是当任何一处达到瓶颈的时候，Queue就会逐渐开始积压IO请求，单个IO的响应时间会随着积压严重程度而越来越高。

19.2.7 缓存管理层

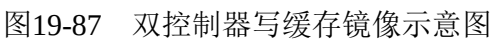
外部存储系统一般都安装有更多的内存来作为数据缓存，并且为了防止突然断电或者系统Down机导致的内存数据丢失，外部存储系统一般都会使用各种方法来保存这些尚未被写入硬盘的Dirty数据。有些直接使用电池给内存供电；有些则使用位于存储设备机架内的微型UPS不间断电源在外部电源故障时将内存中的Dirty数据写入硬盘数据区之后再再将系统Gracefully Shutdown；或者利用内部微型UPS直接将内存中的所有数据Dump到硬盘的固定位置的空余空间之后将系统Shutdown待电源恢复后再读入内存然后写入硬盘数据区；有些则使用电池给内存和一个Flash卡供电，发生电源故障之后，通过某个智能芯片将内存中的数据Dump到Flash卡中存放。

1. 关于缓存的分配

存储系统内存分为两大部分：一是供存储系统的操作系统内核以及

其他上层程序运行所需要的内存空间；二是用于缓存读写数据的数据缓存。数据缓存又可分为读缓存和写缓存，至于读和写缓存所占的比例，不同产品设计也不同，但是一般情况下，写缓存所占比例大于读缓存。因为对于一个大容量的存储系统，缓存的大小相对于磁盘容量来讲是九牛一毛，读缓存在大多数情况下的命中率都不会很高，所以与其将内存分配给读缓存，不如多分配一些给写缓存，因为在Write Back模式下，一般情况下的写总是“命中”的（区别于“写命中”，后文会介绍“写不命中”）。然而请理解，读或者写缓存，并不一定是物理上分界的，各自也并不一定是物理上连续的，任何一个缓存Page只要被标记为Dirty，那么这个Page就属于写缓存，一旦标记被抹掉，那么它就属于读缓存了。一些高端产品具有很大的缓存，这些高端控制器对读写缓存可能会进行物理上的分界，读和写互不干扰。

2. 关于写缓存镜像

对于双控制器架构的存储系统，图19-87 双控制器写缓存镜像示意图

为了防止控制器自身故障所导致的内存数据丢失，两个控制器之间的写缓存是互为镜像的关系，即控制器A收到一个写IO操作，则会立即将这个IO请求及其数据通过控制器间内部互联链路发送给控制器B一份保留，然后才可以向主机客户端返回成功信号。当控制器A将这个IO写入磁盘之后会通知控制器B将它保留的IO副本作废以便腾出空间。如图19-87所示为双控制器写缓存镜像的示意图。缓存镜像的过程会增加主机端的IO响应时间，并且系统的总体写带宽会受制于控制器间的数据链路速率，当然，设计优良的存储系统都会消除这个瓶颈。控制器间链路速率至少要与所有前端总线带宽之和的20%差不多，一般场景下写与读的比例可以按照二八开。

3. 关于缓存命中率

对于连续IO，由于系统预读的效应，命中率通常可以保证比较高。但是对于随机度很大的IO请求，命中率普遍非常低。对于随机IO命中率低的问题目前没有什么特效解决办法，除非使用不需要机械寻道的存储介质比如SSD。被从磁盘读入缓存的数据在传输给主机端之后并不会立即作废，而是停留一段时间来碰一碰运气看看是否一段时间内还有IO请求读取同样的数据或者地址有交集的数据。随着数据被不断的读入，系统总要作废一些老数据而引入一些新数据，存储系统一般使用LRU（Least Recently Used）算法来决定缓存中哪些数据作废哪些继续保留。系统会在Page Table中为每个Page记录一个最后访问时间戳以及Dirty Bit。

预读是显著提高命中率的手段，预读方式根据产品设计而不同，但是目前厂商采用的方法都大同小异。有一类Multi Stream Detection设计需要着重介绍一下。假设主机端有1个进程在对某个LUN发起读IO，IO属性为连续IO，地址从0往上逐1增加，比如LBA0、1、2、3、4、5、.....，此时存储系统会感知到这种连续性从而做大力度的预读动作，缓存命中率非常高；但是如果主机端有两个进程对这个LUN做读IO操作，而且每个进程发起的IO属性也各自都为连续IO，第1个进程从地址0开始逐1增加，第2个进程从地址4开始逐1增加，由于两股IO流是混合发向存储系统的，那么它们的混合排列就可能是类似“0 45 1 6 234 78 5 9”这种方式。存储系统接收到这串IO流，虽然小范围内地址是不连续的、跳跃的，但是大范围之内，地址还是连续的，并且如果将这串被合并的IO流智能地监测分开为两个独立的流（Stream）的话，那么就会发现其实看似随机的IO其实是连续的，那么系统就可以有针对性地加大预读力度，提高命中率了。而不具有这种Multi Stream Detection功能的系统，面对

这种混合流时，其预读力度就不会太大。实际情况中可能存在多个Stream混合，这就对算法的效率有了更高的要求。

然而，对于大块连续读IO来讲，预读就变得没有意义了，此时前端数据的需求处于供不应求状态，所以哪里还会有“预”读这一说呢？所以，有些存储系统提供配置参数，如果用户确定某个LUN所接受的IO都是大块连续IO，那么可以关闭针对这个LUN的预读以绕过系统预读代码流程，节约计算资源。

几乎所有人都认为缓存命中率的概念只对读IO有意义而对于写没有任何意义，其实这种看法是片面的。在某种特殊情况下，写也需要缓存命中的，如果不命中，则需要耗费一些额外的惩罚步骤，说到这里大家可能就有所感觉了。下文将描述所谓真正的“写命中”是什么意思。

4. 关于缓存管理单位

目前所有操作系统对内存的管理都是使用Page为单位，一个Page可以是4KB、8KB、16KB甚至更大。存储系统内部也运行着操作系统，所以存储系统的缓存管理单位也一样是Page。并且，操作系统将文件、块设备等皆映射到Page Cache中，存储系统一样也是将LUN来映射到Page Cache中。既然这样，本章前文中所述的那些在主机端发生惩罚现象，在存储系统内部一样是存在的。如果主机端发向存储系统的IO请求长度不足1个Page，而这个Page在Cache中尚未被读入，那么此时便会触发Page Fault，需要Page In过程。而Page In过程几乎在所有的操作系统中都是一个同步过程，Page只能一个接一个的被从底层读入，而这又更拖慢了整体性能，如果主机端采用的是Write Through模式，那么这个影响会直接被联动到IO源头。

明白了上面这一点，就可以彻底理解所谓“写命中”了。本书前面章节提到的“写命中”只包含了一种情况，即先后多次写IO针对同一地址，则所有这些IO就可以被覆盖为最后所接收到的写IO，最终只需要向下层发起一次IO。而写命中还包含另一个含义，即一旦某个写IO长度小于1个Page，或者不能被Page大小除尽（比如4.5KB），那么除不尽的那部分所落入的Page的内容如果已经位于Cache中，那么此时这个写IO就命中了，可以直接在对应的Page中覆盖入接收到的数据并标记整个Page为Dirty。但是如果除不尽的那部分对应的Page内容并未在Cache中，需要Page In过程，那么此时就说这个写IO未命中，或者部分命中，意味着需要耗费额外的读入过程。不管是否全部命中，只要是IO长度除不尽Page，还会产生额外的数据写入（除不尽的那部分所占的Page需要全部被写入底层介质，而不是仅写入余数部分）。

如果遇到IO起始地址不对齐Page边界的状况，则后果会更加严重。如果某个IO恰好横跨在两个相邻的Page中间，则系统需要读入这两个Page，如果是写IO，则读入之后还要再写入这两个Page。

综上所述，最好使用4KB可除尽的IO长度，即起始地址和长度皆为4KB的倍数，否则会引起性能问题。使用文件系统提供的API一般不会出现4KB不对齐的情况，而如果直接对块设备或者RAW设备作IO，则尽量在IO源头就保证发出的IO符合4KB对齐条件。可见对齐的重要性。

有不少高端存储系统可以对Page的大小进行调整以符合典型的IO长度。比如数据库类程序，它们一般直接使用RAW设备进行IO，而且它们将数据以Extent（或称segment、block等，叫法不重要）的形式存储在LUN中，它们每次读入或者写出的单位也是Extent的整数，此时，存储系统就应当将Page大小调节为与Extent相等即可。如果小于Extent，则会因为更多的Page In请求而影响性能（每个Page In都是同步操作过程），

而且Page越小，Page Table映射表的容量越大，浪费就越多，查找效率也越差；如果Page大于Extent，则可能会产生读写惩罚，得不偿失。所以，使二者相等是最好的办法。

5. 关于Cache分区技术

一些存储系统甚至可以对整个Cache进行分区，为不同的LUN对应的Cache设定不同的Cache大小以及Page大小，这样做就更加灵活了。Cache分区商业化的典型代表是HDS公司的USP以及AMS系列存储系统。如图19-88所示为针对不同应用需求分配不同大小Page尺寸的Cache分区的示意图。

图19-88 为不同应用设置不同属性的Cache分区

如图19-89所示为对分布在SATA盘RAID组中的LUN和分布在FC盘RAID组中的LUN分配不同的Cache分区，进行逻辑隔离。由于SATA盘的低性能，容易造成缓存积压，影响FC盘性能的发挥，所以有必要进行缓存的逻辑隔离，这个问题被称做“快慢盘”问题，随后会介绍。

图19-89 为SATA 盘的RAID 组和FC 盘的RAID 组隔离分区

如图19-90所示为在HDS的存储系统中创建多个Cache分区的界面。图19-91则为分配对应的Cache分区给对应的LUN的界面。

图19-90 设置Cache分区

图19-91 将对应的Cache分区分配给对应的LUN

6. 关于Write Back模式的缓存对读和写IO的影响

大家都知道，在没有缓存的情况下，读总是比写要快的。但是一旦有了缓存，情况就彻底不同了。如果缓存是Write Back模式并且足够大，此时写就可能比读快得多，因为读可能不命中，但是写一定都会“命中”（不考虑写惩罚）。而且Write Back模式的缓存也会弱化连续写与随机写之间的性能差距，因为不管是随机还是连续，主机端发送的所有写请求只要一到达存储系统缓存就被通知成功。判断是否随机和连续是存储系统本身的事情，受影响的也只是存储系统自身后端的性能，但是对主机端来讲没有影响。所以，WB模式下，对于主机端来说，随机写总比命中率低下的随机读要快得多。综上所述，写缓存比读缓存的收益更大，这就是为何写缓存占的比例比读缓存大的原因。

但是，如果遇到诸如每秒上百兆流量的大块连续写IO的情况，那么此时这些写IO数据没有必要占用Cache不放，此时存储系统会触发类似Write Through的直接写盘动作，从而尽量快速地释放Cache空间以迎接后续IO的到来。

存储系统对IO的处理像Linux上的IO Scheduler一样，也会对IO进行Merge处理，一旦遇到地址连续或者有重合交集的情况，那么这些IO会被Merge成一个大IO来提高效率，节约后端资源。

7. 关于写缓存的Flush动作

缓存不可能无限大，理所当然，当写缓存中的Dirty Page达到一定的比例的时候，系统必须要将这些Dirty Page写入对应的LUN以腾出空间来接收后续的IO数据。在中高端存储系统中，这个比例一般都是可调节的。缓存就像一个蓄水池，当蓄水达到一定水位线的时候，就需要有所动作，否则引起水漫金山，后果不可收拾。所以，这个写缓存Dirty比例阈值又被称为“High Watermark”。当水位到达HW之后，引发Flush，

蓄水池放水，水位逐渐降低，当降低到“Low Watermark”时，停止放水。这种Flush触发方式称为“Watermark Flush”，当然不同厂商产品叫法不同。Flush过程毕竟是一个很耗费资源的过程，这期间，主机客户端的IO会受到不小的影响。正因如此，如果追求稳定的写IO性能，则应当适当调低HW，让系统增加Flush的频率，每次Flush耗费的时间将缩短，保持有一定的空余Cache和尽快释放系统资源以用来接收新到的写IO。这样，对于前端就会表现为比较恒定的IO流，而不至于每次都到了最后才收拾烂摊子，导致剩余Cache捉襟见肘，前端会表现为IO流时大时小时快时慢不稳定。

如果系统长时间未达到HW，那么此时系统也需要来一次放水以防止污水存放时间太久，夜长梦多。比如每10秒放一次等。这个阈值一般是不可调的。这种Flush触发方式称为“Time Flush”。

如果系统需要做一些高层的操作，比如做Snapshot等，那么此时务必需要将缓存Flush一次而不管是否达到了时间阈值或者HW。因为只有Dirty Page全部写入底层介质之后，Snapshot才可以反映这个时间点硬盘上的数据。这种Flush触发方式称为“Sync Flush”。

如果遇到大流量的连续大块IO写入，那么HW会频繁的达到，Flush会连续进行，这种Flush方式称为“B2B Flush”，即Back To Back Flush，发生这种情况表明存储系统已经应接不暇了。但是，并不一定非要遇到大流量的IO写入时才会引发B2B Flush，有时候由于后端的瓶颈，也可能引发。比如在系统接收了大量的随机写IO之后，Flush这些随机分散的IO会引起后端磁盘大量的寻道操作，过程会非常慢，而此时前端如果依然有大量的写IO到来，那么HW又会达到，系统将再次触发Flush，Flush也会连续进行，整体性能受到影响。

在系统进行Flush时，Dirty Page被描述为链对象，每个对象包含了一串定量的Dirty Page，每串Dirty Page都会尽量保持地址连续以实现整条写，然后这些对象被传送给RAID层，RAID层判断是否需要读出Parity和被覆盖的数据来进行XOR计算，如果需要则读出，如果不需要则为整条写，则整条数据间进行XOR计算Parity。然后RAID层负责将算好的数据传送给底层设备驱动从而写入对应的硬盘。Flush是一个极其耗费系统资源的过程，这个过程会动员全体资源的支持，尽快地将数据写入磁盘。

8. 关于电池保护缓存

对于一些使用电池保护缓存的产品，当电池发生故障时，比如检测不到电池输入端电压，或者电压降低到阈值之后长时间无法充电恢复其电压，那么此时系统就认为电池失效，并且会将缓存设置为Native Write Through模式，任何写IO必须要写入磁盘之后才会返回成功信号给主机客户端，此时系统性能将会受到很大的影响，写IO进行重排合并优化的时间几乎为0，只能碰运气。当电池恢复之后，又会重新恢复Write Back模式。

有些产品选择在掉电或者以意外宕机之后将缓存中的数据复制到一张Flash卡中存放，比如IBM DS5000系列，这个过程只需要耗费很少的电量，所以只需利用若干大电容来储存电量即可。有些厂商则提供一个小UPS，意外宕机或者掉电之后，将缓存中的数据复制到后端某几块磁盘中存放，这种做法需要更大的电量支持。

9. 关于缓存LUN技术

某些情况下，主机对某个LUN的IO性能要求非常高，容不得半点延

迟，则有些存储系统可以将整个LUN或者LUN的某些部分读入缓存并且对应的Page不会被Page Out，这些数据一直被保留在缓存当中，以获得最小的IO延迟。这种情况下，即便是主机发送大量的随机IO操作，也不会受到磁盘寻道的影响。这种做法的代价就是耗费大量缓存空间。

10. 资源均衡问题

当存储系统中有不同性能的介质层时，慢速介质会消耗更多的缓存和IO资源，因为慢速介质相比于快速介质需要更长的时间才能完成一个IO请求，这样就会导致缓存中对应这个IO操作所保留的资源将要停留更长的时间。这些资源包括Page页面空间、代码堆栈、状态机、Workers进程等。这样，如果系统中同时存在慢速和快速介质，那么快速介质的效果便可能会由于得不到所需的资源而大打折扣，在极端条件下，这些资源可能会被耗尽，导致不管后端采用何种介质，其对外的表现都会处于同一个水平。

此问题可以扩大到任何对资源的争抢问题。比如，即使针对同一种性能层级的两个存储空间，对其下发不同类型的IO请求，也会造成资源争抢。随机IO总是会占据更多的资源，而连续IO本应该是表现出更快的速度，但是可能受累于随机IO的资源争抢，连续IO的效果可能会被拖下水，在极端条件下可能表现出与随机IO相同的速度。

面对这个问题，所有厂商都需要考虑将资源进行合理划分，为高速介质分配一定比例的资源，并且动态调整。根据某些测试结果来看，并不是所有厂商都有此实现的。当混用FC与SATA盘时，某些存储设备针对FC磁盘所表现出来的性能与同样数量的SATA盘所表现出来的性能相当。

11. 存储系统中的QOS

Cache分区、资源预留、自动分级存储等措施，都属于存储系统的QOS范畴，包括上文中所述的解决资源均衡的问题，也属于QOS。不同的主机、不同的应用系统其所要求的IO性能或者容量都是不同的。而一台中高端存储系统往往同时为多台应用主机同时提供存储服务，这就必然要求存储系统对这些主机和应用区分对待，而不是无序的谁抢着算谁的。正像网络设备一样，存储系统也需要利用网络进行数据传输，那么就免不了多数据流争抢资源的问题了，存储系统完全也可以像网络设备那样实现某种加权队列来控制不同数据流的优先级和带宽控制。但是目前这种技术尚未有获得厂商高度推广的迹象。

HDS公司在其高端USP系列存储中提供了多种QOS组件，包括：Cache Partition、Virtual Partition Manager (VPM)、Server Priority Manager (SPM) 和HiCommand QOS Modules (HQM)。缓存分区之前已经介绍过，VPM则是相当于把一台物理的阵列逻辑地划分为多个部分，每个部分之间不争抢资源，整个划分过程通过两层来实现：最底层首先划分缓存以及磁盘，称为Cache Logical Partition (CLPR)，每个CLPR中包含一定数量的缓存以及一定数量的RAID组；第二层为Storage Management Logical Partition (SLPR)，一个SLPR即表现为一个虚拟逻辑阵列，每个SLPR中可以包含多个CLPR。图19-92显示了SLPR与CLPR之间的关系。

图19-92 SLPR与CLPR的关系

HDS公司USP系列存储设备的另一项QOS功能则是Server Priority Manager。该软件的主要功能是在多台主机使用同一存储系统的情况

下，保证高优先级的主机应用的性能，避免其受到其他低优先级主机的影响。可以选择从IOPS或者带宽两个角度来对主机做优先级处理，二者不能同时选择；可以通过阵列端口或者主机FC卡的WWPN来对不同主机进行分类。此外还提供Threshold Control功能，比如当高优先级主机的IO需求下降到某特定值（用户指定）时，系统自动关闭对低优先级主机的IO限制，这样可以避免当高优先级主机的IO负载自己下降到很低程度时，低优先级主机的IO仍然被限制，从而充分利用资源。图19-93所示为在Server Priority Manager中根据端口来配置IOPS或者带宽需求时的界面。

图19-93 Server Priority Manager配置界面

HQM则是针对一系列应用程序（包括Oracle、Sybase、Exchange Server、File Server）所提供的一种与应用紧密结合的端到端的性能监控统计软件模块。以HiCommand QoS for Oracle为例，它的功能主要包括：

- 自动发现Oracle应用的下层构件，并将其展现为拓扑结构；
- 识别、报告Oracle应用与SAN构件的依赖关系；
- 实时监控从应用到存储系统的性能状况，通过历史数据统计提供性能需求及预测；
- 自动发现、监控、分析容量使用情况，协助制定未来容量计划；
- 基于策略，对影响Oracle的存储事件自动做出响应。

类似地，EMC公司的Navisphere Quality of Service Manager（NQM）也提供了类似的QOS功能。NQM首先根据用户的设置对所有进入的IO流量进行分类，可以按照LUN、IO Size、IO类型（读/写）以及未明确指定的所有其他类型IO（Background Class类型）

来划分IO类型，系统将持续监控这些IO类别。之后，系统根据用户所设置的IO优先级信息，包括IOPS、带宽吞吐量、响应时间来对对应类别的IO实现优先级排序从而实现QOS。如图19-94所示为NQM的主配置界面。图19-95所示则为定义IO类别以及限制条件的配置界面。

图19-94 NQM的主配置界面

图19-95 Server Priority Manger配置界面

12. 特殊的I/O和缓存管理

不得不提一下NetApp的WAFL对IO与缓存的管理。（本书多处介绍WAFL并不是笔者对其有崇拜之意，WAFL优势很多，劣势也不少。只是个人比较了解，打算做到知无不言）

大家知道Linux下的EXT3文件系统，其可以使用多种日志模式。第一种是只记录操作行为，而对所操作的目标数据原内容、新内容不记录，这与数据库的日志不同；第二种则是操作行为与目标原内容与新内容都记录，与数据库的记录方式相同。默认方式下使用前者，既降低资源消耗，又能够保证元数据一致性，但是并不能保证文件内容也一致。比如，复制某超大文件到90%的时候，中途断电，重启之后你可能会发现这个文件根本没有出现在目标，或者已经复制的容量只有50%而不是90%。但是如果使用了全内容记录模式的日志，那么重启之后系统只会将原子操作回滚，此时你会发现已经复制过去的文件可能也是90%，也可能是89%，总之回滚粒度会变小，只回滚断电瞬间不一致的已分配间接块，此时这份没复制完的文件你可以删掉然后重新复制。第二种日志

方式会拖慢系统性能，但是却能够获得最佳的一致性以及最小的数据丢失。

WAFL也是一个日志文件系统，也同样使用内容+元数据一起记录的日志方式，这是理所当然的，因为作为一个企业级存储系统来讲，保证数据一致性和不丢失是基线要求。但是如何解决性能拖慢问题呢？WAFL的办法就是将日志直接保存在RAM中而不是硬盘上，再使用后备电池来防止掉电引发的数据丢失，这样，性能问题解决了；然而，容量问题并没解决，WAFL用来保存日志的RAM（也就是NVRAM，其实就是电池保护的普通RAM）只有几GB，最大也不过4GB，除去镜像对方控制器的部分，只有一半，再加上刷盘的High Water Mark被定死为50%，所以每当日志量达到1GB的时候，系统就必须触发刷盘操作。

有人产生疑问了，存储系统中不是有几十GB甚至上百GB的缓存么？为什么不能拿出大部分来充当WAFL日志存放空间？这样WAFL的性能应该更好啊？是的，但是WAFL有个硬指标，也就是至少每隔10s要做一次刷盘动作产生一个一致性点（Checkpoint, CP）。10s的时间并不是空穴来风，有两个原因。WAFL最怕的就是不一致，由于其是一个彻底的文件系统，所以为了保障一致性这个基线要求，其刷盘的间隔相对其他厂商产品来讲是最频繁的，即便有电池保护，WAFL也不能够完全相信电池。这一点从NetApp在磁盘上使用额外空间来存放Checksum的做法就可见一斑，NetApp对底层硬件是完全信不过的，因为它自己不生产硬件，这是其中一个原因。另外一个原因，每次刷盘需要耗费巨大的系统资源，所以需要尽量降低每次刷盘的数据量，如果每次刷盘要几十GB的数据，那系统将会顿卡，前端应用主机的IO延迟将飙升，这是谁都不想看到的。所以NetApp把NVRAM的大小进行了严格限制，这样可以让每次CP的时候能够快速完成同时不至于对前端造成

影响。WAFL这种刷盘方式并不是细水长流型的，是一批一批来的，属于山洪暴发型。

实际中B2B（Back to Back）类型的CP时有发生，也就是系统连续处于CP状态下。特别是高带宽应用下，NVRAM快速充满，造成系统连续CP，此时也相当于细水长流方式的刷盘了。但是由于NVRAM总数据量不大，连续CP不会对前端造成顿卡的影响。

NetApp在系统中除了NVRAM，还配有几十GB的另一部分RAM。这部分RAM主要用来运行操作系统以及各种增值软件功能、存放预读数据以及用于刷盘时的数据读入、修改、写入过程所需要读入的数据（从NVRAM中读出的待写入的内容与RAID校验计算所需从磁盘读出的数据块）。

综上所述，对于NetApp的产品，其实际等效写缓存为NVRAM容量的四分之一，目前最大也就是 $4 \div 4 = 1\text{GB}$ 。至于系统剩余的其他RAM的大小会对性能有多大影响，尚不可乱猜，但是从商务角度考虑，厂商间都在飙规格、飙参数，各家产品都不会在RAM规格上落后，管他有多少效果，先配到业界水平再说。同时，不了解技术细节的用户也时常被厂商所忽悠误导了标书。

本书的一个目的就是让所有人了解存储底层细节。

19.2.8 数据前处理和后处理层

外部智能存储系统之所以称为智能，并不是由于其能管理大量的磁盘，生成大量的LUN，拥有众多的前后端接口，而本质原因其实是它们拥有一些高附加值的Data Cooker。这些Data Cooker专门对原始数据进行预处理或者后处理，它们就像大厨一样对原始数据进行加工处理，最后

实现一些让人耳目一新的功能，包括：Snapshot、LUN Mirror、LUN Clone、Data Migrating、Data Tier、Disaster Recovery、CDP、Deduplication、Space Reclaiming、Virtualization等。

这些Data Cooker会以各种方式嵌入到整个存储系统中，比如嵌入到系统底层驱动链中，或者直接以用户态程序存在。它们的插入会改变原本的数据流路径甚至数据内容。比如Snapshot，一旦针对某个LUN生成了Snapshot，那么任何针对这个LUN的IO就都会被Snapshot模块过滤和处理。Snapshot属于前处理，即数据在被写入介质之前就会被处理。后台Deduplication就属于一种后处理，即当数据被写入介质之后，当系统不忙的时候，Deduplication模块对数据进行采集和计算并且消除重复的数据块。后处理的例子还包括比如Space Reclaiming，这个模块也是需要后台来完成对浪费的存储空间的回收。

关于数据前处理和后处理的内容已经在本书之前的章节中介绍过，这里就不再多说了。

19.2.9 存储系统处理一个IO的一般典型流程

下面用一张图来结束本节。如图19-96所示为一个外部存储系统处理一个IO请求的典型流程图。图中所示的仅为基本的流程，并未引入具体的细节，细节过程可以参考本节之前的文字。

图19-96 存储系统IO请求典型流程图

19.3 IO性能问题诊断总论

本章前两节已经遍历了从IO起源地到IO目的地之间的全部过程。本节将对其做一个总结。

业务层，始作俑者！IO的源头是应用程序，而程序是为了业务需求而生的。程序对IO性能的需求直接反映了业务的类型和复杂度。如果我们能够精简业务，简化业务流程，提高业务效率，那么将会从根本上降低程序对IO性能甚至整个IT系统的需求。

应用层，身不由己！人在江湖，身不由己！程序反映的是业务逻辑，业务逻辑有多复杂，程序就有多复杂，相对应的数据库在查询或者更改数据的时候所发出的IO就有多复杂。当然，程序发起的IO操作并不是与业务逻辑复杂度严格对应的。一些应用程序或者数据库执行脚本对底层IO欠考虑，编程的时候粗枝大叶，并没有调查底层的情况，所以并未做到优化处理。有时候应用层只要稍微变一变设计、语句执行顺序或者IO调用方式等，就会带来很大的收益。所以对IO要求比较高的应用程序在设计时需要专门对IO模块进行设计。比如SQL语句，实现同样的业务需求，换一种方式，也许就能得到成百上千倍的速度提升。

操作系统，主战场总指挥官！操作系统平台提供了所有程序的运行环境，直接操作底层硬件，提供多种IO调用模式以满足不同需求。在Windows下，IO Manager使用IRP作为信使来将所有驱动链连接了起来；在Linux下，系统使用bio作为信使也将各个驱动层联系了起来。

物理磁盘，鞠躬尽瘁死而后已！存储系统中最受尊敬者：物理磁盘！它们勇往直前，前仆后继，从不偷懒，永远勤劳工作于第一线！存

储系统最受伤的劳动者：物理磁盘！为什么受伤的总是我，为什么如此辛劳最后却还是成为众矢之的？只怪我独臂难挡重兵啊！

网络传输层，我怎么感觉不到你！底层链路层是很容易被忽视的一个地方，不过好在这一层出问题或者出现配置不当导致的瓶颈的几率相对其他层次来说是很低的，只要链路速率匹配，线缆接触良好，一般都不会引发性能问题。

19.3.1 所谓“优化”的含义

本章前半部分对系统IO路径进行了解剖，形成了一张IO路径图，并且洞察了IO请求在整个系统路径中的流向以及每个层次对IO的处理方式，并且引出了一些需要注意的地方，可以这么说，所谓“优化”，其实应该是一个消除瓶颈和问题的过程，如果各处都没有瓶颈或者问题，那么优化是没有意义的。整个主机和存储系统就像生物体一样，是一个造化产物，其本来就具有完美的自我优化功能，这是生命之本能，但是总会有一些邪恶势力入侵导致系统功能紊乱，或者一些原发的系统自身功能紊乱。我们需要做的，就是消除这些导致功能紊乱的问题，让系统恢复如初。

诊断可以在两个层次上进行，一层是在主机操作系统内核或者存储系统内核设计的时候所做的优化，即原生的优化，这种优化不但要考虑普遍情况，还需要考虑特殊情况，也就是说需要提供足够的参数设置让用户根据不同的IO属性和要求来调整这些参数。第二是在主机操作系统或者存储系统中进行瓶颈探测和消除的工作，这种工作为表面优化，即尽最大所能让系统原有的能力发挥出来。表面优化又有两层，第一是做到参数的设置与系统设计思想以及当前的IO属性相匹配而不是背道而驰；第二是做到平衡负载、按需分配（注意，不是负载均衡，平衡不等

于平均），将资源最大化平衡地利用。

这便是性能优化的含义了。性能优化既不是魔术也不是魔法，它不可能让不存在的东西变得无中生有（但可能将有的东西弄丢），对于生物体来讲，可以服用一些药物来暂时性的激发各器官功能，但是这样做是有害的，损害器官功能，它违背了自然和谐的本质，并且也不能维持长久。对于计算机系统来讲，可以使用超频等手段来提升一点点性能，但是这无疑是自欺欺人。要做到本质上的优化，只能从造物时的设计入手，不同生物体有着不同的设计，有的力大无穷，有的可以飞行，存储系统也一样，不可能让老虎长出翅膀。我们能够做的只是治病救人，而不是去改造人，改造是造物主的事情。

性能优化的本质包含四个字：合适合理。比如用一个低性能的存储系统来承载一个对性能要求很高的应用，而又要求进行性能优化，妄想让存储系统“超水平发挥”，这是不可能的。能做的只有将各种参数调节成与这个存储系统底层设计相合适的值，而且合理设置分配存储资源，好钢用在刀刃上，榨干整个系统性能。这样做了之后，合适合理就完成了，那么性能优化也就完成了，如果结果不理想，那只能说明存储系统整体性能不能满足应用需求，需要根据情况扩充硬盘数量或者更换更强大的存储系统。

19.3.2 如何发现系统症状

当然，下药须对症，而要判断症状就需要望闻问切。对于存储系统，一样需要一番望闻问切过程。要分析路径中哪里出了问题，就一定要在整条路径中都来进行跟踪。

对于用户程序所发起的IO，Windows下的Filemon和Linux下的Strace

是跟踪程序发起的IO的好工具。当IO进入内核层之后，在Windows下可以使用性能监视器，Linux下则可以使用iostats等，AIX系统下这方面的工具更多，比如topas、filemon、nmon等。

利用这些工具来查看IO在内核层是否被分解或者合并，是否产生了读写惩罚，产生的惩罚本身是读还是写等，在这一层都可以判断出来。而当IO出了主机操作系统内核进入外部存储系统控制器之后，就必须在存储控制器端使用IO监测工具来探测了。理论上讲，主机端所监测到的内核层IO，应该与外部存储端所接收到的IO一一对应，但是并不排除中途的智能网络传输设备会对IO进行诸如合并、分解、复制等操作。一般情况下，非智能网络设备并不会改动任何数据IO，所以主机内核层的下部发起的IO其实也可以通过外部存储系统上的监测工具来查看。

IO流入外部存储系统内核之后所经历的处理过程，也可以使用存储设备所提供的监测工具来查看，一直到IO流出后端适配器最终靶向目标磁盘之前的IO行为都可以被监测到。最后，通过分析这整条路径中所有层次上的IO监测数据，就可以得出大致的症状。

提示： 不到万不得已，最好别使用工具来直接检测应用程序发起的IO，因为这些IO监测工具一般是第三方开发并且都是插入驱动链中来作用的，弄不好会对程序产生影响，更甚至可能引起系统崩溃。所以，高层的监测工具是最后的救命稻草。

19.3.3 六剂良药治愈IO性能低下

由于本章前面的部分已经对IO路径有了详细的分析，相信大家在阅读了之后都会对如何优化IO性能有了自己的宏图，所以本节就不再详细介绍，只做一下大方面的总结，总结出六剂治愈IO性能低下的良药。面

对大多数莫名其妙的性能低下问题，只要对症下药，那么十之八九都会药到病除。

症状1：存储系统每秒接收到的**IO**数远未达到系统标称值，链路带宽也远未达到，前端接口的**Queue Length**远小于**Queue Depth**，并没有严重积压现象，存储系统后端磁盘繁忙度很低。主机端程序的**IO**延迟很低，但是吞吐量以及**IOPS**并未满足要求。

可能病因：阳火不旺，不能自举。调用方式欠火候。

病因判断：**IO**源头的并发度不够，程序使用了单线程同步**IO**。在这种情况下，程序无法利用全部的存储系统性能，虽然每个**IO**的延迟很低，同步调用的程序也会获得一定的性能，但是毕竟没有异步调用或者多线程同步调用时所显现的性能。

药方：生发阳气。修改程序使用异步**IO**调用或者多线程设计。

症状2：存储系统每秒接收到的**IO**数远未达到系统标称值，链路带宽也远未达到，前端接口的**Queue Length**值接近**Queue Depth**值，显示有积压的**IO**，存储系统后端磁盘繁忙度很低。主机端程序的**IO**延迟很低，但是吞吐量以及**IOPS**并未满足要求。

可能病因：经络不通，气滞淤阻。**Queue Depth**过低。

病因判断：可以判断此时程序发起**IO**的并发度依然不够，因为程序的**IO**延迟很低，如果是并发的**IO**造成底部积压，那么对于程序来讲其**IO**的响应时间会很高而不是低。正因为程序发起的**IO**是一个接一个的，所以每个**IO**的响应时间才会保持较低的值，但是底部的**Queue**依然造成了积压，由于存储系统标称值远未达到，后端也不繁忙，所以这种积压一

定不是由于存储系统后端磁盘资源或者计算资源耗尽导致的。所以问题就在于Queue Depth不够。Queue Depth有两处可调节，一是主机端存储控制器驱动程序处，二是存储系统前端接口适配器驱动处。

药方：疏通经络，通则不痛。调节以上两处Queue Depth，使其增大到一个合适的值，根据本章前文所述的Queue Depth的一些规则来确定。另外，本病例中如果主机端程序的IO延迟很高，其他症状不变，那么就表明程序可能在使用异步IO或者多线程并发同步IO，治疗方法相同。

症状3：存储系统检测到**Cache Hit**率极低，磁盘几乎百分百繁忙，而前端**IOPS**和带宽很低，业务所要求的响应时间无法满足。

可能病因：急火攻心，阴虚火旺。IO随机读过大。

病因判断：IO的随机度过大，严重增加了存储系统磁盘寻道时间，性能跟不上。

药方：降火凉心，滋阴养阴。向RAID组内加入更多磁盘，或者更换高转速高规格的磁盘，或者使用缓存LUN等类似技术，或者直接将对应磁盘更换为SSD等存储介质，滋阴养阴，此为治标。改善程序的IO设计或者降低业务的复杂度，也就降低了IO的随机度，降火凉心，此为治本。

症状4：从主机端或者存储系统端监测到的主机内核发出的IO从数量和容量上都表现为较高的值，通过计算理论上可以满足程序的需求，但是程序层却表现得很迟缓，IO延迟、IOPS、带宽等皆达不到从存储端监测到的数据所体现的指标，那么此时可以判断在主机内核中产生了惩罚。

可能病因：心神不宁，心烦意乱。主机端发生惩罚现象。

病因判断：此症为心病，判断起来非常难，因为不管是从主机端内核底部还是从存储端表现出来的IO行为是无法判断是否在主机内核处产生了惩罚的。唯一判断方法是在用户程序下方使用IO跟踪工具来监测本层的IO，然后与内核层发出的IO做对比，如果发现不匹配，比如底层的IO在数量和容量上为程序发起IO的数倍，则可能引发了惩罚。但是对于多程序并发访问外部存储的情况下，很难判断是哪个程序引发的惩罚，此时可以使用差量法判断：在一个IO流恒定的时间范围内，让程序提升一定倍数的IO并发度或者IO发送速率，同时监测底层发出的IO情况，不引发惩罚的程序对底层的IO浮动影响绝对值较小，这个浮动值基本等于程序层的浮动值，而引发惩罚的程序会引起底层IO浮动上升一个较大的值。利用这一点可以初步判断到底是哪个程序引发了惩罚，之后，可以单独部署这个程序进一步调查其IO惩罚情况，从而做出改正措施。

药方：静心安神。对于主机端内部产生惩罚，导致的原因有两种。

第1种是因为程序发起的IO Size不等于Page Size的整数倍或者IO Size平均值远小于Page Size。第1种情况可以通过更改Page大小来解决，不过修改Page大小一定要经过深思熟虑。因为OS内的Page是全局的，如果只是某个程序的IO Size过小，则不能因噎废食而影响了其他程序。如果程序的IO Size虽不为Page整数倍但是规则，比如512B、1KB等，则尽量使用DIO模式来越过Page Cache。但是一定要注意DIO的对象，如果是文件或者块设备，那么在某些OS之下（比如AIX），这些目标对象依然需要使用Page Cache，而只是越过了文件系统Page Cache，并未越过块设备的Page Cache。所以在这类系统下，使用DIO的时候最好是针对字符设备，也就是RAW设备。此外，也并不一定非要使用DIO或者RAW，如果程序偏要使用内核缓存来IO文件或者块设备，那么务必在

调用时不要使用Write Through参数，因为这样会加重惩罚力度。如果程序发起的IO Size并不是512B对齐的，而又因为惩罚而产生了性能问题，那么此时解决办法也是尽量不要使用Write Through模式来进行IO调用。

第2种是因为程序发起的IO的起始地址并没有落在Page的边界，导致额外的读入或者写出1个或者多个Page。对于第2种情况，应当尽量去修改应用程序，别无他法。总之，修改应用程序让其发出的IO Size即为Page的倍数，起始地址又落在Page边界，这是最理想的状况，此为治本。

症状5：主机端并未发出大量的IO请求，但是存储系统却忙得不可开交，同时存储系统上并没有诸如Deduplication等Data Cooker的后台操作，主机端IO延迟偏高。

可能病因：心神不宁，心烦意乱。存储端发生惩罚现象。

病因判断：这种症状属于更加隐秘的心病，是最难判断的一种。这种症状只是一个极端的状况，实际中很少会有惩罚严重到通过监测数据就可以观察到的地步，加之多主机共同访问存储系统造成的混合因素，更加难以判断。但是一些存储系统会对这种内部惩罚加以监控和输出，用户只需要根据输出值来判断惩罚的力度即可。造成存储系统内部发生惩罚的原因除了主机端的两种原因之外，还有第3种，即LUN起始地址的不对齐（见本章前文）。一旦发生这种情况，纵使主机端的IO再规则和对齐，那么存储端也会出现惩罚现象。

药方：静心安神。对于LUN不对齐引发的惩罚，多半是因为在创建LUN的时候选择了不匹配的操作系统类型，此时则需要做LUN迁移动作，或者删掉重建。对于主机端IO Size不能被Page整除的情况，可以说

这种情况对于程序发起的IO是比较常见的，但是对于主机内核发起的IO，一般很少见。程序发起的不规则IO经过主机内核之后一般都会经过Page Cache，所以会被映射为规则的Page对齐的IO之后被发向存储系统处理。但是不排除一些特殊情况，比如程序使用了DIO，或者直接操作RAW设备，同时又没有让IO Size为存储端Page的整数倍，这就比较常见了。对于这种情况的解决办法，需要综合考虑。存储端的Page Size一般是可以调节的，如果存储端的Page Size与主机端的平均IO Size相差太大的话，那么需要考虑将存储端Page Size设置为所有主机端IO Size的平均值。对于主机端发送的IO起始地址与存储端Page边界不重合的情况，也是很少见的，如果排除了LUN不对齐导致的因素，问题依然存在，那么只能说明主机端不按常理出牌，这种情况必须要修改应用程序的IO设计。

症状6：存储系统单个**RAID**组中硬盘很多，转速也很快，存储系统控制器处理能力也强大，这个**RAID**组中有多个**LUN**分配给了多个主机使用。就是不知道为什么，当一台主机访问其上的**LUN**时，性能很好，但是一旦有另一台主机同时访问这个**LUN**，或者访问这个**RAID**组上其他**LUN**的时候，性能骤降，两台主机获得的性能之和还不如之前单台主机所获得的性能。

可能病因：筋骨劳损，不堪重负。多**LUN**共享同一**RAID**组IO冲突。

病因判断：磁盘最怕什么呢？寻道。一个**RAID**组中的磁盘数量是一定的，这也就注定了它能够提供的性能是个定量，如果将多个**LUN**放在同一个**RAID**组之上，那么这多个**LUN**就要来瓜分整个**RAID**组的性能，分配的方法是有讲究的。正如本章前文所述，**LUN**在**RAID**组内的分布有多种方式，可以按照纵向顺序的分布，也可以按照横向横跨所有

磁盘分布，可以混合型分布，甚至可以完全随机分布。

简单纵向顺序分布的LUN，由于每个LUN只占用1块或者几块硬盘（根据LUN大小和磁盘大小而定），所以每个LUN的性能都是固定的，不同的LUN之间不会相互影响，除非它们占用的磁盘有相交。但是这样的分布方式也注定了每个LUN的性能有限，如果使用146GB硬盘，而LUN的大小小于146GB，那么这个LUN所拥有的性能容量只等于单块磁盘的性能。

简单横向顺序分布的LUN，由于每个LUN都横跨了所有硬盘，所以每个LUN都能享受到整个RAID组的性能容量，但前提是一段时期内只针对其中1个LUN的IO被处理。如果一旦其上的多个LUN都在接受IO，那么由于每个LUN是横向排布在所有硬盘上的，同时读写为不同的LUN中的数据，就要求磁盘不断地寻道，磁头臂时而摆动到LUN1的区域，时而摆动到LUN2的区域。这就好比一个十字路口，任意时刻只能有一条路的车辆处于行进状态，另一条路车辆必须等待，一个绿灯周期结束的时候，另一条路的车辆开始发动，发动过程是很慢的，就类似磁盘寻道过程。如果遇到特殊堵车情况，可能一个绿灯周期只能过1辆车。所以，由于十字路口的存在，造成了两条路的车辆都无法维持高速行驶，单位时间内两条路的行进里程之和远小于没有红绿灯时的正常行进里程。

所以说，被寻道打断的滋味是极其难受的，存储系统内部在面对这种情况的时候，或许会做一些特殊的考虑。比如类似Linux下的IO Scheduler中的CFQ的思想，将所有LUN平等地对待，并且做一个比较合理的周期，如内核可以依次从每个LUN的Queue中取出一定量的IO充入后端适配器驱动的Queue中，每次充入的IO数就决定了每个LUN得到服务的周期。这个周期是有考究的，就像红绿灯一样，如果你设定红绿灯

周期为10s，那可能有些车还没启动利索呢，红灯已经到了。相应地，如果内核依次轮流地从每个LUN的Queue中只拿出1个IO充入到底层Queue中，那么底层磁盘所做的动作就是每执行一个IO，就寻道一次将磁头定位到很远处的下一个LUN的区域，执行完后再寻道，每个IO都需要寻道，一段时间内，大跨度寻道所耗费的时隙比例远远大于传输，那其效率可想而知。所以这个周期需要根据实测结果和理论推导一同确定。同一个RAID组内越多的LUN同时接受IO，那么整体性能就越差，磁盘寻道造成的浪费比例也就越大，整体性能差到无可救药。

注意： 以上所描述的性能损失只针对连续地址IO，因为连续IO之所以本应该性能良好是因为寻道开销很小，而多LUN横向分布导致了针对每个LUN的连续IO整体表现为了随机IO，所以性能会大打折扣（也就是前文所述的“将原本有的东西弄丢”）。而原本针对每个LUN的随机IO却并不会打折，因为随机IO本来就需要频繁寻道，所以多LUN并发的随机IO，和单个LUN的随机IO相比，在磁盘数量相等的时候，并不会损失这一组磁盘所提供的整体性能，每个LUN获得的性能之和接近于RAID组整体性能。但是多LUN相对于单LUN来讲，每个LUN所获得的性能是下降了，但是整体来讲却没有浪费。

对于分布式RAID模式的LUN排列以及类文件系统模式的LUN排列方式，其本质与上面所述的两种方式类似，只不过是上述两种方式的一种结合后的产物。所以这里就不再过多分析这两种模式对性能的影响了。

另外，不管何种方式分布的LUN，如果同一个主机上有多个程序同时访问这个LUN，并且如果每个程序各自访问LUN的不同区域，那么这些程序就会瓜分这个LUN所拥有的性能容量。比如，有两个程序，一个

在对某个LUN做连续IO，而另外一个则对同一个LUN做随机IO，那么连续IO所获得的性能将会被严重拉低。只要引入了磁盘寻道的任何操作都会将连续IO性能严重拉低，甚至拉低到与随机IO一个等级上。

症状样例：某视频监控系统，用户**200**路摄像头数据同时以**1Mb/s**的速度通过千兆网写入视频主机，主机和存储设备通过**IP-SAN**（千兆）连接，存储的卷是**3**个**14**块**1TB**的**SATA**盘组成的**RAID 5**组，共**42**块盘，**3**个**RAID 5**组，然后**3**组卷直接组合成一个大磁盘再分配给主机。这时候直接在主机上打开视频文件的速度很慢，慢的时候需要**10**多秒甚至无法响应。

症状分析：把所有盘组成一个大盘，所有文件都分布在所有磁盘上，那么小范围顺序IO就会变为整体性随机IO，性能显然是无法保证的。**200**路至少对应**200**个文件，同时对**200**个文件进行IO，整体来讲就是一种随机IO，本身速度就是很慢的。再加上随后的读入文件进行播放的操作，读+写+随机，这就更加重了磁盘的负担。解决办法是将文件分开存放，最好是每个文件只占用一个物理盘，然而这样做需要修改应用程序，临时的做法只能是增加存储端的条带深度，增加到最大，即便这样做也不能从根本上解决问题。

药方：放松筋骨，按需平衡负载。根据具体需求，确定使用何种LUN分布方式。比如，如果不计成本，则完全可以为每个LUN单独创建一个有足够性能容量的RAID组，并且主机端程序所操作的文件也尽量放到不同的LUN中，做到从上到下都是独享一组底层磁盘的性能容量，这样做是最好不过的，也是最完美的。但是实际中不可能不考虑成本，所以每个RAID组上一般都会有多个LUN存在，那么此时就需要进一步规划：是采用纵向分布，还是横向分布，或者还是自定义比例的纵横结合的分布。可惜的是，正如本章前文所述，目前市场上的产品能否做到

让用户来选择如何分布LUN的产品几乎没有，大多数产品都使用横向LUN分布方式，所以用户只能在这种分布方式上来做文章了。

而如上文所述，横向分布的LUN在面对多LUN并发的连续IO方面可谓出尽洋相，性能差得一塌糊涂。但是总归还是有一些死马当做活马医的缓解的办法。解决的办法：将RAID组的Stripe Depth，也就是Segment，调整到所允许的最大值。这样做就会加大每个Segment所包含的连续地址范围，使得单块磁盘包含更长的连续地址段，所以，每个IO所占用的磁盘数量也将会随之减少。

不过通常情况下，由于大多数产品可供配置的最大Stripe有限，假设为256KB，如图19-97所示。如果是8块数据盘，那么Stripe Depth就相应为32KB，如果每个IO平均为32KB大小，那么这个IO就只占用这块磁盘。但是如果主机端使用异步并发IO调用方式，那么单位时间会有多个IO同时被这组磁盘执行。假如8个32KB的连续地址IO同时执行，那么正好就占用了8块磁盘，另一个LUN在这个过程中只能等待接下来的磁盘寻道至自己的区域从而为自己执行IO操作。

如果将Stripe Depth进一步加大，比如256KB，也就是说Stripe宽度为2048KB，如图19-98所示。如果此时IO还是并发8个32KB，那么这8个IO其实都落入同一个Segment，只需要底层针对对应的物理磁盘的一次IO即可读出或者写入这8×32KB的数据，而其他7块磁盘都是空闲的。如果还有额外7个LUN，那么如果碰巧的话，每个LUN均可以同时被执行8个32KB的IO，每个LUN都会得到不错的性能。如果将Stripe Depth继续增大，那么到了最后其实就相当于纵向分布的LUN了，整个LUN都分布在一块硬盘上（硬盘足够大），到了这时候，每个LUN之间就是真的井水不犯河水了，但同时每个LUN的性能容量最大值也就等于单块磁盘的性能容量了。其实，增加Stripe Depth的做法正是从RAID 3→RAID

4→RAID 5的进化路线中关键的一步，Stripe Depth越大，并发几率也就越高，对于LUN分布来讲，也一样是这个思想。

图19-97 Stripe Depth=Segment=32KB

图19-98 Stripe Depth=Segment=256KB

当然了，以上的解决方法只能是乱放枪碰运气，要从根本解决，还需要目前的存储产品设计上更加人性化，提供更多可设置的LUN分布方式了。当然，最终的治本的方法，还是要淘汰机械硬盘，使用无机械寻道的存储介质。

到此，6个典型症状已经介绍完毕了。实际中所遇到的情况很有可能是以上几个症状并存，或者由一个症状引发了另一种症状的连锁反应，使得问题的解决更加复杂化，但是只要掌握了每种症状的治疗方法，再加上仔细的判断和分析，相信总会药到病除。

19.3.4 面向SSD的IO处理过程优化

SSD使用的NAND Flash是个特殊的介质，本书前面章节也充分描述过。SSD所表现出来的性能确实强劲，那么当SSD被用在外置磁盘阵列中，与机械硬盘混用时，而且多块SSD再次被组成RAID组时，这样性能又会得到加成。有没有一些针对SSD的优化措施呢？有不少，有些恐怕我们之前压根没想到过有什么相关性的东西，可能都与SSD有关。

- 元数据结构优化：由于无须考虑IO的重排等动作，针对SSD RAID组的内存中元数据结构可以大大简化，提高执行效率。
- WB可以改WT了：由于SSD响应时间很快，所以没有必要先缓存在珍贵的RAM中，然后再刷盘了，直接透写到SSD中即可，让更多的RAM空间用于缓存机械硬盘的脏数据。SSD盘片自身有大容

量的RAM用来缓存脏数据，自身可以对数据进行合并操作，并且使用超级电容来防止掉电，所以可以放心的WT到SSD。SSD对进入的写IO属于WB模式，因此先WB到阵列全局RAM还是直接WB到SSD中的RAM，效果相差不大，所以直接WB（对于阵列控制器来讲是WT到SSD）到SSD中即可。

- 不用双控镜像了：由于针对SSD RAID组对于阵列来讲是WT操作，所以阵列的双控制器之间不需要对WT的数据进行缓存镜像操作，节约了镜像通道带宽，更重要的是降低了写IO延迟。
- Queue可以加大了：见上文。
- 阵列控制器需要支持Trim了：由于SSD机制决定，最好能对SSD做Trim操作以提高性能。但是SSD插到阵列中之后，不直接面对主机操作系统了，它们之间相隔了一层阵列控制器，所以此时如果依然要支持Trim，那么就需要直接面向主机操作系统的阵列控制器支持Trim，阵列控制器将Trim指令再次根据SSD RAID组的地址映射信息转换为针对每个SSD的翻译映射之后的Trim指令然后下发到每个SSD中，一个主机Trim指令会被转换为多个Trim指令。

19.4 小结：再论机器世界与人类世界

各位，请醒来，我们现在又回到人类世界了。我们所看到的这些机器，只不过就是一台台黑漆漆的落在那儿，不会动不会说话。但是刚才的那场机器世界的游历，我们发现机器世界是如此美妙，每一步都精确无比，每一步都井井有条，鬼斧神工般地运行着。

第20章 腾云驾雾——大话云存储

■ 云

■ IAAS、PAAS、SAAS

■ 集群与虚拟化

■ 云计算、云存储、云服务

■ 计算、传输、存储

ERP、ITIL、Cloud，这是21世纪伊始从西方发达国家兴起的次时代IT新概念。如今，ERP并未获得广泛的推广和应用，ITIL仍是空中楼阁，Cloud又出来了。实话实说，没有经济腾飞催生大批信息化发展到一定程度的企业，没有迫切的需求，没有达到一定的积累和高度，这些特别高层的IT管理运营方法论是很难获得推行的。

而Cloud这两年刚刚兴起并又从国外炒到了国内，弄的是乌烟瘴气，众说纷纭，你说你的，我说我的，各地争相搞云。搞起来之后却好像鲜有人买账。搞云是要来满足用户需求的，而不是为了搞而去搞，后者背后有一定的不良因素推动。

然而，Cloud与ERP和ITIL还是有一些区别的。云是直接为盈利而催生的，相对于前两者来讲还是显得比较实在。前两者属于一种管理者角色，务虚；而云属于一种执行者角色，务实。虚的东西落地非常难，而实的东西很容易落地，这也是云为何快速兴起的原因。有人头疼了，说

我怎么每次听到人讲云，就发现很虚幻呢？那是因为给你讲云的那人自己也虚，是带有一定目的来忽悠的，而不是来给你传道授业的。本章，笔者将会站在中立角度为大家通俗演绎到底云是什么。

20.1 太始之初——“云”的由来

目前，云计算、云存储、云备份等云技术可谓是铺天盖地地袭来。这其中不乏有一些浑水摸鱼者，其本身并不具备多少云性质，却也在打着云的旗号想在市场炒作中分一杯羹。目前市场对一款产品是否为云产品并没有一个明显的界定，因为云这个东西本身的定义就没有一个标准，各种机构纷纭其说，莫衷一是。好在SNIA（存储网络工业协会）在不久前发布了CDMI（Cloud Data Management Interface，云数据管理接口）标准用来规范在云系统中实现数据传输、存储和管理的一系列规范，这个标准只是针对一个云系统中的数据存储和管理部分而制定的接口标准。对于云系统中的其他部分，比如用户接口、硬件管理接口等，目前并没有一个标准来遵循。

正因如此，判断一个产品是否为云产品，也没有什么标准可循。

说道“云”这个词，其由来显得有些不可思议。我们都知道微软Office软件中的PowerPoint，其中有一个图形，如图20-1所示。我们在制作PPT的时候，经常会用到这个云状的图形来指代一堆网络设备、存储设备或者服务器设备。

图20-1 云一词的由来

然而，在国内好像人们都习惯把这一堆设备称之为“网络设备群”、“服务器群”，也就是用“群”这个字，起因是因为国外在指代这堆设备的时候，一般都是用“Cluster”这个词。而Cluster中文一般被翻译为“簇”或者“集群”，所以“群”就这么出现了。某一天，也是国外某人在讲授某PPT的时候，顺口说了一句比如“The servers in the

cloud”，“Cloud”一词就这样诞生了。至于到底是谁第一次说出这个概念的，之后通过什么途径流行开的，又是谁率先用云来指代大规模IT基础设施的，尚无从考证。

云诞生并且被公之于众之后，人们对云基本上产生了4种理解：云即设备、云即集群、云即IT系统、云即服务。

20.1.1 观点1：云即设备

这种观点也是最原始的观点，它仅仅指代某一堆设备，也就是PPT中的云状图形所囊括的那些设备。此时，云不具有任何意义，仅仅是一个指代词，为了表达方便而已。这种对云的认知目前依然存在，一些对云不太了解的人，往往一开始都是这样认为云的。然而，你不能就说这种认知是错误的。这个认知只是云发展的一个阶段，也确实是云的组成部分，因为没有设备支撑，哪来的云呢？任何概念都要有实实在在的支撑。

20.1.2 观点2：云即集群

在第一个观点，也就是“云即设备”的基础上，人们的认知开始逐渐发展，如果云仅仅是一堆设备的话，那么这堆设备也只是一堆设备而已，那么云这个概念也就就此罢了了，没有什么发展了。然而事物总是要不断向前发展，给自己开拓新道路。一堆设备肯定不是这个事物发展的尽头，如果再向前进一步会变为什么呢？

当然是集群了，这就像多个人组成社会一样，一堆人放在一起便会自行发展成社会，一堆设备放在一起，其高级形态就是一个集群。设备之间是有机联系起来的，共同协作的。那么云在这个认知和发展阶段，

就表现为一个集群，不管是计算集群还是存储集群。

20.1.3 观点3：云即IT系统

然而，一个集群能做什么呢？集群只是一个有机结合可以写作的设备集合，那么如果云只表示到这一层的话，显然也是没有任何生命力的。云要发展，就必须再往上走。集群之上是什么呢？赋予一堆硬件灵魂的，是软件。软件和硬件组合起来才是一个完整的有机系统，也就是云即IT系统，比如某企业的IT系统，就是一朵云；某运营商的IT系统，也是一朵云。对云的认知发展到这一层，就快要露出本来面目了。

20.1.4 观点4：云即服务

那么，云即IT系统这个观点还能继续再发展么？还可以的。问一问，IT系统是用来做什么的呢？

答1：“运行程序用的。”

答2：“支撑和服务于企业生产的。”

答3：“用来盈利的。”

这三种答案，一个比一个高级。第一个答案显然太过技术化，没有实际直接意义；第二个则很靠谱，企业运营过程中需要借助IT系统来做支撑和服务；第三个则更加激进了，直接使用IT系统来盈利，暗指这家企业靠的就是出售或者出租其IT系统。

我们着重来看第三个回答。如果想直接拿某个IT系统来盈利，换了你，你会怎么来盈利，采取什么样的盈利模式呢？

答1：“直接卖掉，拿钱，多省事！”

答2：“我搞出租，把整个系统租给别人用，我收租！”

答3：“我开发个网络游戏，然后用这个IT系统来运营这个游戏，谁想玩的就交钱！”

咱们来分析一下这三种盈利方式，第一种最直截了当，盈利最快，直接卖掉数钱去了，但是他再也无法用这个IT系统赚钱了；第二种则稍微聪明点，虽然盈利慢，但是资源还是掌握在他手里；第三种则非常精明，但是需要前期很大的投入来开发游戏，然而一旦成功运营，所获得的收益将是巨大的。

比较一下这三种方式，第一种受众面将会很小，因为需要购买一整个IT系统的人几乎很少有；第二种受众面稍大一些，因为需要租用服务器租用存储空间的人还是比较多的，但是也非常有限；第三种呢，受众面则非常之大，几乎所有人都是潜在的受众，因为所有人都会对游戏娱乐有基本需求。受众面直接影响到利润。

题外话： 笔者个人极度讨厌网游，尤其是最近充斥互联网的黄色网游，属于一种精神毒品，本人坚决抵制网游。有空学学谷歌地球、百度地图，搞点真正能产生社会价值的业务，而不要像国内某运营商一样搞些虚拟垃圾来俘虏一大批人，浪费电，浪费劳动力，全民玩虚拟网游，这样下去很危险。

话说回来，这第三种盈利方式，就属于一种用服务来赚钱的模式，提供游戏服务。还有众多服务表现形式，比如邮箱、网页、博客、音乐、影视等。这些都是大众最基本的需求，所以盈利面很大。在一套IT系统之上可以同时提供多种服务，那么可以最终认为，能够提供某种形

式IT服务的一整套IT系统，这就是云。

从这个角度上来讲，所有的互联网运营商，比如各大网站，全部都是云运营商，它们后台的IT系统从建立之初就已经有了云化的性质了，但是离真正的云还稍有差别。

20.1.5 云目前最主流的定义

可以从上文中体会到，设备组成集群，集群组成IT系统，IT系统用来服务。这一串的理解恰恰就是对IT系统运营的理解。那么是否可以将这四个理解组合起来，形成一个最终的观点？云是一个可运营的IT系统。但是，这个定义少了某些关键的东西，就是资源迅速灵活地部署和回收。也就是说，云当前的主流定义是：一个可运营的、迅速灵活部署和回收资源的智能IT系统。云为何会有这个关键点？下一节中会给出答案。

笔者对目前业界普遍被承认的云性质做了总结。云应当具有如下性质：云必须体现为一种服务交易而不是实物交易；云提供商拥有一定规模的硬件基础（比如足够的网络以及服务器和存储设备）；云提供商对客户提供一种资源的租用服务而不是资源本身的易主买卖。

凡是具备以上三个特性的系统，都可以称其为云系统，或者云服务。比如一些域名公司出租的网页空间，其包含了如下服务：服务器硬盘上的空间租用、Web服务程序运行资源租用、网络带宽流量租用、域名维护服务。这些服务到期将自动回收，比如网页空间到期不续费，那么服务器有权将其网站内容删除，所以这不是一个易主买卖，所有的资源都归供应商所有，你得到的只是服务而已。

对于“域名供应商也是云提供商”这个说法，可能有些人并不同意，

因为他们认为只有一些高端的、让人望尘莫及的大型运营商，比如 Amazon、Google 等所提供的服务才是云服务。他们还认为只有底层使用了硬件集群和虚拟化技术的系统，才具备最基本的云资格，这些观点都是比较狭隘的。

20.2 混沌初开——是谁催生了云

20.2.1 一切皆以需求为导向

任何事物的出现、发展状态、衰落，都有其底层因素。云这个词从PPT中被衍生出来，如果没有肥沃的土壤，那么它充其量也只能乖乖地被用来指代“一堆设备”了，或许根本不会为人所知。这些肥沃的土壤，就是日益壮大的用户需求。“互联网时代的用户需求”，如果把这个当做一个课题来讲的话，那真的是永远也讲不完，太多了！互联网以及接入终端（PC、手机等）的广泛普及，使得大众的交流渠道产生质的改变，以前通过书信或者电话才能传达的信息，现在可以通过互联网用各种形式来实现了，比如传统的语音、图像视频、论坛、E-mail、博客等，之前大众之间的大量交流需求被一下子爆发了出来。

这种需求的爆发，其直接效应就是信息爆炸，IT系统的底层支撑设施快速被饱和。就像一个被大量人口所充斥的城市一样，其交通系统（网络）、教育医疗等公共服务机构（各种应用服务器）、基本的饮食供应（电力供应）以及住房（存储系统）等就会被大量饱和，导致拥堵不堪，效率低下。

面对互联网时代的需求爆炸，传统的IT系统已经显得无法满足。为何呢？看看传统IT系统是怎么运作的。比如，某运营商市场部门分析出未来一年内网页游戏业务将会有20%的增长，而目前支撑网页游戏的IT子系统的利用率已经接近100%，需要对现有的系统进行扩容，包括增加Web Server节点的数量、增加数据库服务器节点的数量以及扩大存储系统的容量。系统扩容就需要采购新设备，需要遵循一系列流程，耗费的周期很长，甚至已经可能慢于业务的变化周期。也就是说可能一个

业务当你忙活着部署的这段周期内，市场需求可能慢慢消失了，部署业务的速度慢于市场需求变化的速度。此外，技术层面还存在停机的风险。

而这个运营商的另外一项业务——在线视频聊天室，由于经营惨淡，支撑这项业务的IT子系统利用率不足60%，有40%的余量没有被充分利用。如果能够将这40%的余量用于支撑网页游戏业务的扩充需求，那么就是最好不过的了。但是这样做在技术层面风险很大，如果将两套业务系统部署在同一个操作系统中，会大大增加两种业务的粘合度，不利于后期的运维管理；另外，将同一个业务分布在两个资源孤岛上，更加不利于维护。最后，由于整个数据中心的服务器与存储设备繁多，各种协议、各种不同厂商的设备混存，架构复杂，这种情况下，单靠手动来部署、管理和回收各种资源已经变得非常有挑战性了，一是效率低还容易出错，二是速度慢到可能影响业务上线，尤其是在一个承载多项业务的数据中心中，各种业务对底层的要求都不同。

以上这个场景可以总结为以下3个问题：

- 业务部署周期太长；
- 资源不能充分回收利用，资源孤岛林立；
- 手动部署已经无法满足要求。

这3个问题已经成为所有互联网运营商的痛点。如何解决？请继续阅读下面的章节。

20.2.2 云对外表现为一种商业模式

笔者恰恰不这么认为，或者说，上面这些观点只属于狭义范围内的对云的定义。而广义范围内的云，其实并不局限于硬件或者软件的技术

或者架构，最初的、广义上的云，其实是一种商业模式，而当商业模式与具体的计算机技术相结合之后，便产生了云这个代名词。所以说，云既是一种商业模式的指代，又是一种计算机技术的大集合。

正因为云本质上起源于一种商业模式而不是技术模式，那么云当然也就没有一个外在的像技术一样严格的标准了。为何这么说呢？举个例子，有10个人，每人都想开一家餐馆，那么其结果一定是这10个人开的餐馆，每个都是不同的。首先就是店面布局不同，其次是价格不同，再次是管理方式和管理所用的工具不同，最后就是菜肴种类和口味也不同。那么我们回到计算机领域，比如还是这10个人，每人都有10台服务器，现在想让这10个人用这10台服务器来获取最高的利润，其结果肯定也是不同的。比如有人选择直接卖掉它们，易主交易；有人则想到了另外的方法，比如有人做起了网站空间出租业务，还附带一系列的诸如域名、网盘等其他业务；也有人在其上开发了一个软件平台，做起了文档阅读服务；还有人则利用这些服务器开办了计算业务。选择出租空间的，就可以叫它云存储；而选择出租计算资源的，就可以叫它云计算了。

云虽然没有一个外在标准，但是基于同一种商业模式所诞生的不同产品，它们显然是有共性的，比如都拥有自己的硬件基础架构，包括网络、服务器、存储设备和软件平台；都提供特定的用户接口，比如通过网页方式来享受服务，或者通过客户端程序连接到云供应商处来获取资源等。

这种共性不代表标准，也无法被标准化。自古到今从来没有人或者机构对其他人的商业模式来制定标准，就像并没有人规定，所有的餐馆必须是两层楼，必须以某某规格来装修一样。对于装修简陋的餐馆，它依然还是一家餐馆，你不能因为它简陋就认为它不是餐馆了。正如有些

域名商只用一台服务器就承载了多个网站一样，虽然只有一台服务器，但是它的商业模式已经是云模式了。

云商业模式中又可以细分为多种具体的商业模式，比如IAAS、PAAS或SAAS等。这就好比有些餐馆只卖炒菜，而有些只卖面条一样。关于这三个服务模式，下文会有介绍。

20.3 落地生根——以需求为导向的系统架构变化

20.3.1 云对内表现为一种技术架构

正如20.2.1节中的那个场景，传统IT系统的技术架构已经对商业需求产生了制约效应。而解决那两个痛点的技术手段，自然而然的非虚拟化莫属。服务器虚拟化，即虚拟机系统，充分地利用了资源，辅以诸如VMware虚拟机系统中的VMotion、DRS（Distributed Resource Scheduler）等技术，极大地增加了部署灵活性和资源均衡性。如果那个运营商已经部署了虚拟机系统，那么之前那个痛点就自然解决了，旧业务的余量将会被自动回收（对物理服务器的资源消耗降低），新业务所需的应用主机可以直接以虚拟机的方式被部署在物理机上，可以与旧业务使用同一台物理机，但是操作系统却是各用各的，避免了粘合影响。另外，部署虚拟机比部署物理机所耗费的时间大大减少，极大地提升了针对新业务的响应速度。最后，使用一种资源自动化分配与回收平台来解决自动化部署的问题。这样，那3个痛点就这样被轻松地解决了。

传统的数据中心“太硬”，需要在其上增加一个弹性层，让其“变软”，成为软数据中心或者称其弹性数据中心。而实现弹性软化的一个方法就是使用虚拟化技术，包括计算资源和存储资源的虚拟化。

大家知道，古代的马车，近代的汽车，都是使用实心轮子，这样是无法跑快和跑稳的。后来人们发明了充气轮胎，最终使得现代的汽车可以高速行驶，并且保证平稳。对于一个数据中心也是一样的，早期，没有任何虚拟化措施，买了多少台服务器，就是多少台，不能变多也不能变少，当然，除非你关掉几台；同样，你买了多少存储就有了多少空间，这些空间不会变多也不会变少。那时候，人们根本想象不到，如今

可以利用虚拟机技术将一台服务器虚拟化为几十台虚拟机，也不会想到Thin Provision以及Deduplication技术可以让存储空间像变魔术一样从小变大或者从大变小，正如古代人的飞天梦想如今早已实现一样。如果说硬数据中心是早期的实心轮胎的话，那么软数据中心就是充气轮胎了。如图20-2所示，在传统的数据中心硬核心之外，被包裹了一层软外壳层，增加了弹性，这层软外壳可看作是虚拟机管理系统、分布式文件系统、集群/Scale-Out架构的NAS/SAN，以及Thin、Dedup等将硬计算与存储资源进行虚拟化和灵活化处理的层次，这一层形成之后，整个数据中心就变为了软数据中心了。再向上走一步，如果还能做到部署回收自动化、可度量化、服务化、可运营的数据中心，那么这也就是个云数据中心了。

图20-2 给数据中心增加弹性

再比如之前用服务器办计算业务的那个例子，他只有10台服务器，如果来找他买计算服务的人越来越多，已经超过10个客户了，那么第11个客户就只能等待么？不能等，否则他会找其他供应商的。那怎么办？他自然而然地想到了虚拟化技术。他在这10台服务器上部署了虚拟机平台，每台物理机器可以虚拟出多台虚拟机器，这样他就可以接更多的订单了。再之后，他发现每次客户都是通过FTP方式来上传需要计算的数据，而每次他收到之后都要手动转换格式然后载入计算，为了解决这个问题，他开发了一种自动转换格式的程序，而且还做了一个网页接口，客户每次只要通过网页方式上传，并选择对应的参数，那么服务器会自动把这些数据转换成对应的格式，自动计算，并且最后将计算的结果自动通知客户，大大节省了人力成本，人所要做的工作只是管理和收费即可。再后来，连收费系统都变成了自动的，利用网络支付平台实现了自动支付，其服务器的规模也日益增大，最后发展成为一个底层为大规模

集群的、中层通过虚拟化技术抽象虚拟化的、表层全自动业务处理的大型云计算系统。

对于云存储，也是一样的道理。从最简单的存储空间租用，到最后变成底层大容量的存储设备集群、中层加入存储虚拟化层以及各种数据管理功能层（Thin、分级、快照、容灾等）、表层实现全自动业务处理的云系统。

综上所述，虚拟化和集群化是云系统中两个重要的角色。另外，云系统中还需要另外一个重要的角色，也就是一个负责资源自动部署、调度、分配和回收的管理者角色，它表现为一套软件模块，这个模块对内与整个云中的各个资源部分通信以达到对资源的管理，对外则负责响应业务部署的需求，将这些需求转化为对内的资源调度分配和管理。这个模块综合来讲，就是“自动化”。

用合适的技术架构来承载互联网时代的商业需求，云对内表现为一种技术架构。集群化、虚拟化、自动化是作为一个云来讲所必需的特性，然而，有了这些还不够。一个云想要达到可运营的状态，还必须做到可度量化，任何用户使用了何种资源，为期多长时间，耗费多少成本，毛利率几何，报价几何，这些都要经过精确的度量、定价过程。

20.3.2 云到底是模式还是技术

云到底表示一种商业模式，还是表示为一种技术架构呢？可以说这是个鸡生蛋和蛋生鸡的问题。前者是先有了云的思想，然后才在对应的技术架构上来实现了这种思想；而后者则是先有了大量的物质基础，有了对应的技术架构，比如虚拟化和集群架构，然后自然想到了如何利用这些物质基础来获取最大利润，于是便催生了云这种商业模式。我们已

经很难追溯云的发展史了，所以到底是谁催生了谁，可能永远也说不清，道不明了。但是有一点是肯定的，两者结合之后，一定是相互催生，相辅相成，一直到今天被炒的如此火热的程度。

图20-3 数据的存储、管理与运营

大家看一下图20-3。最早期的时候，存储系统只注重数据存储，只给你提供一块空间，数据怎么管，怎么用，底层存储不关心。后来随着人们需求的增长，这种心态已经完全落伍了。存储系统开始注重数据管理，针对各种需求开发了各种数据管理功能，比如数据保险，也就是快照或者CDP，重删以及Thin，数据容灾、克隆、迁移等。再后来，存储系统除了关心怎么存、怎么管，还开始关心怎么用的问题了，也就牵扯到数据运营了。存储系统管得越来越多，越来越上层了，越往上走，就已经不是存储系统这个子系统所能掌控的了，此时需要贴近用户的应用，会注重业务展现，针对传统存储厂商来讲是个很大的挑战。由于互联网日益蓬勃发展，越来越多拥有IT资源的机构都想通过互联网来进行运营，包括ISV、NSP、传统电信运营商，这就注定了云的发展。

有人认为云的本质是虚拟化技术或者集群技术，这一点也比较偏颇。应当说云的本质是一种由虚拟化和集群技术支撑的以服务为模式的可运营的IT系统，也就是商业模式与技术架构共同组成了云系统。云只是利用了很多技术来实现商业模式和目的而已。云包含的技术不仅仅是虚拟化和集群技术，还有其他各种技术，比如并行计算技术等。可以说只要存在的技术，都可以融入到云中，但是虚拟化和集群技术是大规模云所必需的。

云就是一个可以提供某种模式服务的、可以根据业务迅速响应并且自动地、迅速而灵活地部署和回收资源的智能IT系统。至于到底要多么

智能才算是云，没有一个固定标准。可以这么说，传统的数据中心如果叫它云1.0。那么目前兴建的这些融入了新技术的可灵活部署的数据中心可以叫做云2.0。

如图20-4所示为从用户需求到最终业务展现的流程示意图。这张图概括了之前章节所表达的所有观点。

图20-4 基于需求的云进程

20.3.3 公有云和私有云

此时我们已经有了一个云数据中心了，那么这个数据中心是打算只对企业内部开放服务，还是对互联网之外的任何人开放服务呢？有时候某个企业的私有数据中心想转变为云模式，只对企业内部提供云服务，比如存储空间申请、桌面备份和恢复，以及企业内部应用系统的快速部署等，那么这个云就属于私有云的范畴。而如果某个数据中心如果想要对外营业，通过互联网提供各种云服务，那么这个云就属于公有云的范畴了。其实叫做公用云和私用云更准确，用户并非拥有这个云而只是在使用这个云。公有云数据中心一般都属于运营商，运营商更懂得运营之道。

1. 私有云让企业IT部门的角色彻底转变

对于一个企业来讲，其IT部门的角色往往是比较尴尬的，甚至对于一个搞IT的公司也是这样。企业的IT部门在传统观念下被定义为一个底层支撑部门，但却基本上没有自己的话语权，话语权被掌握在业务部门手中。比如某制造企业需要上一个新产品生产线，这个项目对IT部门的要求则是提供足够性能的服务器、存储等以用来承载新业务系统。IT部

门在接到这个任务之后，就必须进行调研之后发起招标采购等流程。我们看看这个过程中存在的一些特征和问题。

首先，IT部门在整个过程中始终处于业务部门的牵引之下，一切围绕业务需求来开展，所有的采购、经费申请等必须以业务需求为前提，业务没有需求就基本上申请不到大批经费和资源。

其次，传统的IT架构中，不同的业务一般不会运行在同一台物理服务器中，这就导致每上一个业务，基本上就要采购一批新设备。而如果之前某个业务下线了，或者企业根据市场状况决定将某业务缩量生产偃旗息鼓，那么此时IT系统中用来支撑这些业务的设备负载就会随之降低，释放了资源，但是这些被释放的资源却得不到利用。

基于上面两个因素，IT部门有潜在的意愿来将自己的角色进行彻底转变，之前被业务部门牵着走，现在它想提高自己的地位，不说跑到业务前面去，也要与业务部门处于平等地位。怎么实现这种角色转变呢？

“服务”这两个字恰好满足了这种转变的需求。比如，IT部门采取了一系列措施将自己彻底打造为一个以服务为导向的部门，这些措施包括：建立规范的资源申请流程，不管是从日常桌面维护还是新业务上线方面，建立电子工单审批系统，任何人想要获得任何IT服务都必须填写工单，审批之后IT部门输出对应的服务；另外还建立了一套可度量的资源使用记录统计系统，比如可以统计某其他部门在某段时间内使用了多少IT资源，这些资源等价于多少成本。

这套东西被推行之后，IT部门这个独立服务角色就定型了。既然是服务，那么其就有了一定的话语权，并不是“用户”（企业其他业务部门）说什么IT部门就要干什么的。这样也就在向企业申请各种资源方面

有了更大的弹性和主动权。这样，业务部门在向IT部门申请IT资源的时候就是以一种协商态度而不是强势的牵制态度了。另外，由于资源的使用变得可度量，IT部门就会随时掌握整个企业对IT部门资源的使用程度，并做出合理的预测，在申请后续经费等资源时变得更有说服力。

好，既然IT部门已经成为一个独立的可服务型的部门了，那么下一步就势必要考虑到它自身运营的成本问题。最大的成本来自于哪里呢？

当然是设备购置成本。上文中所述的那个资源得不到充分利用的场景，已经成为最令IT部门头疼的问题，迫切需要技术手段来解决，而解决这个问题的最佳技术就是虚拟机技术，这也是为何目前越来越多的企业打算部署虚拟化IT环境与虚拟桌面环境的原因之一。另外，IT数据中心还有更多一系列的其他问题，比如扩容费用高昂、扩容维护停机、迁移困难等，而这些都有对应的技术手段解决，比如使用Scale-Out集群与虚拟化技术等。总之，集群和虚拟化（包括虚拟存储与虚拟计算）这两大技术手段可以为数据中心解决很多棘手的问题。引入虚拟化与集群之后，IT数据中心将变得更加有弹性，比如原本规划的时候，考虑一台物理设备可以承载100台虚拟机，那么承载105台是否可以？某些场景下可能也没有问题。这样，IT部门在申请建设资金时也就更加有弹性了。

是什么可以让IT部门地位提升？它想提升就提升了么？没有这样的事情。本质是因为现代企业越来越依靠IT，任何事情都离不开IT部门，所以它的地位自然就会升高，到了一定的程度，加上一些促发因素比如虚拟化、集群技术的支撑，那么IT部门角色转型也就顺理成章了。

2. 共有云受制于互联网网络带宽发展受限

想让某个云数据中心提供服务，就要满足一个基本条件，即网络带

宽要够。网络带宽直接限制了一个数据中心能够提供的服务的种类和级别。如果一个云数据中心是面向企业内部提供服务的，即私有云，那么就不必担忧网络带宽的问题，因为企业内部的网络带宽是非常充足的，其可以支撑任何种类和层面（IaaS、PaaS、SaaS）的服务，比如IaaS，比如用户可以直接申请一块存储空间而使用对应的协议（比如iSCSI、NFS、CIFS等）进行挂载使用，速度也不慢；再比如用户可以申请一台虚拟机，通过远程桌面来登录使用，可以上传各种应用程序并安装运行。

而对于共有云，问题就来了。如今互联网接入带宽偏低，除了像日韩等国之外，其他国家宽带接入速率远未达到可以承载IaaS服务的可容忍程度。大量的用户还是通过1Mb/s或者2Mb/s的宽带来接入。这种只有100KB/s或者200KB/s吞吐量的接入速度，你让用户购买什么服务呢？给他个iSCSI协议访问的存储空间是不现实的。充其量提供一些诸如网盘之类的上传下载服务，网页服务就不用说了，1Mb/s基本上已经够用。视频服务呢？标清码率的视频2Mb/s速率勉强可以承载。总之，常用的SaaS服务（网页、聊天、视频、网盘、在线游戏、基于Web的信息管理系统等）基本上可以承载于低速网络上。大部分PaaS服务也可以通过互联网来提供，比如微软的Azure等。而IaaS就有些困难了，比如你购买了一台虚拟机的使用权，运营商给了你一个虚拟桌面连接地址，此时1Mb/s速率基本上很勉强，如果你要想在其上安装一些应用程序的话，那基本上很慢，因为你得先把安装包上传上去。所以有些IaaS提供商一般都会预先装好必要的软件比如数据库、中间件等，打包在虚拟机磁盘映像中，根据用户的选择将对应的映像制作成虚拟机。另外，SaaS有着更广泛的用户基群和更长的历史，而PaaS与IaaS是伴随着云的兴起而逐渐被引出的概念，SaaS会借助云的兴起而获得持续发展，PaaS和IaaS则会随着云而加速发展。

综上所述，云目前最能够被广泛推进的地方就是新建的数据中心，几乎所有新建的数据中心都会被向云方面引导和建设，不管是企业数据中心还是运营商数据中心。企业兴建私有云数据中心，而运营商则兴建混合云数据中心，也就是同时对内和对外服务的云数据中心。

20.4 拨云见日——云系统架构及其组成部分

如图20-5所示为一个云数据中心系统中的关键层次。经过前文的描述，大家此时应该对云的由来、发展和表现形式，云能干什么，有什么特点，由什么技术堆叠而成等方面有了一个框架式的了解了。这一节带领大家再深入一层，从概念深挖到具体的架构层。

图20-5 云基础架构层次

20.4.1 物理支撑层

云是一种方法，但是它也必须承载于物质之上。对于一个云数据中心来讲，供电、散热、安防和维护等都是必不可少的，这些元素构成了云的最底层，也就是物理支撑层。

20.4.2 基础IT架构层

数据中心一定要有支撑其运营的IT系统设备，包括基本的网络、服务器和存储系统，还需要一个网络/设备管理系统。然而，这些服务器与存储设备并不是一个个的孤岛。上文中说过，集群、虚拟化、自动化，是一个大规模的纯正的云所应当具有的技术特征。

云底层的集群架构可以有两种实现方式；第一种途径是利用现有的网络，网络技术是把现有的、零散的、非专用的所有资源整合起来，在其上加入虚拟化层，形成一个计算和存储的分布式集群，在这个集群之上再来实现云服务层，第二种途径是专门的集中式并行集群数据中心，加上虚拟层，然后再加上云服务层。但是一般新兴建的数据中心都是采用后者，也就是直接使用专用的集群系统来充当基础IT架构层。

说明： 利用大规模网络来实现云计算的例子也是有的，最为成功的一个莫过于SETI@Home寻找外星人项目，利用Internet上的计算机为其计算。这个程序作为一个屏幕保护程序存在，每当屏保激活时，程序便连接服务器端来获取一段需要计算的数据，然后开始计算，并且将结果存储在本地。每次屏保被激活，程序便开始计算，直到算出结果为止，将结果传输到服务器端然后再次下载需要计算的数据来计算。这样，整个互联网中的电脑就组成了一个超大规模的网络集群。而这个场景确实非常真实和准确地反映了“云计算”这三个字。

这似乎又矛盾起来了，网络计算是一种分布式计算，而云又应当是一个集中式计算提供者，仔细想来其实不矛盾，这就是一种虚拟化的表现，即先用网络计算整合所有计算资源，然后再用虚拟化的方式将这些资源出售或者出租。

20.4.3 基础架构/集群管理层

有了集群还不够，还必须在这个集群之上覆盖一层或者几层虚拟化层来增加整个系统的弹性，将所有资源虚拟化为资源池。对于计算资源，也就是集群中的服务器节点，通过使用VMware、Citrix等虚拟机平台可以完成这个工作。而对于存储节点呢？也需要有这么一种虚拟化平台，而目前来看，能够满足这种需求的存储空间虚拟化平台，只有分布式文件系统或者分布式卷管理系统才能满足。

另外，网络、服务器以及存储集群基础架构需要一个管理模块，负责整个集群的监控、硬件资产管理、硬件故障更换管理等。

20.4.4 资源部署层

有了基本的网络、服务器和存储集群，还是远远不够的，需要一个用来管理和驱动这个集群的角色。上文中说过，集群硬件之上是虚拟化的弹性包裹层，比如VMware的Vsphere4（计算资源包裹）以及分布式/集群文件系统或者分布式/集群卷管理系统（存储资源包裹）。利用Vsphere所提供的VMotion与DRS（Distributed Resource Scheduler），可以将虚拟机在集群节点中灵活移动，而且可以做到资源动态分配与回收。然而，一个云数据中心中并不一定只有一种虚拟层，可以有多种不同种类的虚拟层，这样，就需要一个独立的虚拟机与虚拟存储资源调度分配软件模块，它通过调用这些虚拟化模块所提供的接口来完成整体的资源调度与分配回收。

20.4.5 中间件层

当有了物理环境、IT基础架构、基础架构监控管理、资源分配部署回收层之后，一个充实的基座就有了。在这个基座之上，就可以完成各种业务的部署了。应用层与资源层之间可能需要一个中间层来适配。这个层次不仅位于应用引擎和资源部署引擎之间，云架构中所有层次之间可能都需要各种适配。

20.4.6 应用引擎层

应用引擎层则是产生各种业务应用的温床了，这一层提供一个通用的业务开发平台，或者将其他平台所开发的应用适配进来，然后统一发布。

20.4.7 业务展现与运营层

现在万事俱备，只欠东风了。数据中心的硬件、软件、架构都已经

被打造为集群化、虚拟化和自动化的形态，各种业务也可以随时部署和撤销，底层资源得到最大化的利用，降本增效。那么还欠缺什么呢？刚才所列出的这些，都只是对你自己有意义的事情，是为了让你自己更好地去适应这个市场，适应不断膨胀的用户需求。那么这些东西对用户来讲，没有任何意义。用户不关心底层用不用集群或者虚拟化，更不关心底层是人工部署还是自动部署和管理。用户只关心他能得到最快的服务与响应，更关心提供服务的方式、界面、操作便捷性、展现的如何、收费是否合理等。对于一个云来讲，业务展现于运营层是最终关系到这个云盈利模式及利润的关键。

云服务是让数据中心实现盈利的另外一种商业模式，说白了，就是卖数据中心，把数据中心的所有资源整合起来，虚拟化，然后再分配，再以租用和服务的方式出租。大规模的云之内必须要有虚拟化层，一是用来榨干物理设备的资源，二是用来整合成大的资源池，如果没有虚拟化，则将无法管理一个个的孤岛，资源分配的灵活性达不到要求。

我们可以把云分为云存储、云主机、云计算这三大块服务。IaaS属于云存储或者云主机范畴，PaaS和SaaS则属于云计算范畴。

笔者总结了一个云服务架构简图，如图20-6所示。

图20-6 IaaS、PaaS和SaaS

出租数据中心可以在下面的几个层次上进行。

1. 基础设施即服务（IaaS）

卖存储空间，卖虚拟机，卖计算服务。所谓Infrastructure As A Service，或称硬件即服务（Hardware as a Service, HAAS），其中的I就

代表了云系统中的硬件设施，诸如服务器、网络、存储系统等。如上图所示，云提供商的存储集群和计算集群，皆可以出租。可以只卖存储，也可以只卖计算，或者买计算的赠一定容量的存储空间，当然笔者在此只是开放地设想而已。IaaS并不算是云计算的一种，因为IaaS并不直接负责用户的计算，而只是提供硬件平台，具体计算的细节由用户自行部署。

1) 如何卖存储空间

可以有多种方式，如卖裸空间、卖文件存储空间等。裸空间就是说最终用户的操作系统看到的是一块硬盘，所用到的协议当然首选iSCSI方式，以便于跨越IP网络，即用户终端通过iSCSI Initiator连接云提供商处的iSCSI Target从而获得一个或者几个LUN。文件存储空间方式又包含两种方式：一个是基于传统NAS协议的访问方式，另一种是基于HTTP协议或者服务商专用的其他协议访问方式。基于NAS协议的存储空间需要使用NFS或者CIFS协议来挂载服务商处的Volume，基于HTTP方式访问则是更高一层的实现方式，比如网络硬盘等大部分都是用HTTP方式。基于裸磁盘和NAS协议的Volume来存储数据，在操作系统层面具有一定的通用性，可以实现各种应用程序透明访问分配给它的空间。而基于HTTP协议的访问方式，不具通用性，只在定制的情况下才使用。

对于云中的存储系统，诸如Thin Provision、Deduplication、Dynamic Tiering等特性应该说是必须的。云的一个作用就是高效和成本降低，Thin与Dedup这两种数据缩减技术可以降低不必要的存储空间占用；而动态分级则可以进一步节省存储成本。

2) 如何卖服务器/虚拟机

这里说到的依然是IaaS，即卖的是基础设施而不是更高层级别的服务。服务器资源该如何出租呢？总不可能卖给用户一整台物理服务器吧，那样就没有任何意义了，用户很大可能根本用不到这台物理服务器所能提供的最大资源。

然而，更不可能把一台服务器分割成几半来出租，除非有一种虚拟化的方式，将物理服务器虚拟成多台虚拟服务器。的确是这样的，这正是云服务中的服务器资源的虚拟和出租方式。提供商通过某种虚拟化解决方案比如Vmware、Citrix Xen等。比如用户需要1台运行Linux系统的DB2数据库服务器，1台运行Windows 2003系统的Exchange服务器，2台运行Windows 2003系统的Web服务器，每台服务器需要3个网络接口，则提供商通过某种图形界面生成这个配置，然后发送到云服务管理端，云自动在现有的虚拟机平台上按照配置要求创建好对应的4台虚拟机。至于这些虚拟机最后是不是落在同一台物理机器上，需要根据更多的因素来决定了，况且虚拟机可以动态地在物理机器之间迁移，资源也可以按需分配，这些在技术上都可以实现。生成的虚拟机，可以给用户提供一种方式，比如Telnet、SSH或者远程桌面等管理方式像在本地管理一样来管理用户从云提供商处购得的虚拟机，在其上安装用户自己需要的操作系统和软件等。而服务器的硬件维护、供电、网络等则可以全部交由云提供商解决。

虚拟机平台需要考虑的几个功能，一是动态迁移，即虚拟机可以不影响应用系统而在物理机器之间迁移，二是强劲的资源动态分配调度，三是管理方便。

Amazon在IaaS方面提供了两个产品：弹性计算云（Elastic Compute Cloud, EC2）和简单存储服务（Simple Storage Service, S3），分别对应了主机计算集群和存储集群。除了Amazon，提供IAAS服务的还有

3tera、GoGrid、Rackspace Mosso和Joyent等。

2. 平台即服务（PaaS）

卖中间件服务，卖软件平台服务，卖开发定制服务。相对于IaaS，PaaS则屏蔽掉而且不出租Infrastructure，转而出租更高一层的软件平台，在这个平台上，用户可以制作并测试符合自己要求的网络应用程序。在对应厂商提供的PaaS平台上开发的应用程序一般只能在这个厂商的云基础架构中运行，也就是说，PaaS是一个孕育各种应用程序的平台，但是这些应用程序又只能在当初孕育它的平台上运行。PaaS属于一种云计算服务，因为这个平台是一种运行于硬件集群上的软件，用户租用了这个平台其实就等于租用了计算业务。目前几个比较知名的PaaS平台有Windows Azure、Force.com、Google AppEngine、Zoho和Facebook等。

3. 软件即服务（SaaS）

卖内容，卖结果。SaaS是云服务中的最外层业务。云提供商直接向用户出售业务级别的内容，而与业务相关的数据计算，都在云内部完成。SaaS是目前互联网上非常普遍的一种服务，比如我们最常用的Web网页服务、QQ等及时聊天服务，都属于SaaS的范畴。目前比较知名的大型的企业级SaaS提供商有Epicor、NetSuite、Salesforce.com和Zoho的客户关系管理（CRM），SAP Business ByDesign和Workday的ERP套件等。

至此，我们就清晰地看到了一个数据中心是如何被分层出租的。值得说明的是，PaaS提供商可以租用IaaS，而SaaS提供商也可以租用IaaS和PaaS，从而实现不同层次的逻辑分割和耦合。

20.5 真相大白——实例说云

在我们冲出这团云之前，笔者想用两个具体的实例来向大家展示一个具体的IaaS提供商到底是如何向用户提供IaaS服务的。读者在看完下面的一些具体细节之后，会彻底理解云服务的本质。

20.5.1 3Tera Applogic

3tera公司通过两种方式来提供云服务：一是直接提供公共云IaaS服务；第二种是将云平台软件授权给第三方，第三方在自己的Infrastructure上部署云服务平台。日本老牌的电信运营商KDDI，就将其部分业务迁移到了3Tera的Applogic云中运行。KDDI并非租用3Tera的公共云，而是直接购买了3Tera提供的Applogic云虚拟化层，将其部署在自己的硬件Infrastructure上，形成了自己的云系统对内或者对外提供IaaS服务；同时，在IaaS之上再使用自己的PaaS和SaaS平台层为其用户提供PaaS和SaaS云服务。

3Tera的IaaS平台名为“Applogic”。Applogic是一个可以实现IaaS功能的软件虚拟化平台，或者按照官方的说法，是一个网格操作系统。咱们拒绝一上来就忽悠一些摸不着边的东西（往往就是因为这样才让人晕的），还是先来看一下它的底层架构，然后再说明这种架构可以带来什么样的变革。先来看一张图，如图20-7所示。这张图就是整个Applogic系统的总体架构图，共分4个大层次。

图20-7 Applogic架构简图

1. 第一层：硬件层

本层包括主机服务器、存储系统（SAN和DAS均可）、网络设施。值得一提的是，Applogic并不要求底层存储系统必须是基于SAN架构，其可以用本地IDE或者SATA硬盘来作为存储空间。若干主机服务器通过千兆以太网连接起来，形成一个集群，或者叫它网格也可以。我们就是要在这些有限并且分配很不灵活的资源池之上，实现一种管理方便、使用方便、资源分配灵活的虚拟化层。

2. 第二层：分布式核心虚拟化层（Applogic OS）

本层由3个子层组成，分别是DVM、GVS和LCM层。

- 对计算资源的虚拟：Distributed Virtual Machine Manager（DVM）子层

本层的核心是虚拟机技术，即在物理主机上通过Hypervisor引擎来虚拟化成多个虚拟的主机。大家都知道Vmware的ESX server就是这些技术。只不过Applogic使用的是Xen的虚拟机平台。

- 对于存储资源的虚拟：Global Volume Store（GVS）子层

Applogic使用的是自研的分布式文件系统，在这个文件系统之上虚拟出Volume，Volume可以是Mirror、Clone、Snapshot等。每个Volume都在多个物理主机上有镜像以解决HA问题，并可以提升读性能。这些Volume对于最终的虚拟机来说就是裸磁盘。在这个虚拟层之上，每个节点将自己的本地存储空间贡献出来，所有节点的存储空间被整合起来并虚拟化，再分配。

- 对网络资源的虚拟：Logical Connection Manager（LCM）子层

对此没什么可多说的，就是将物理网络搞成虚拟网络，虚拟机都有的技术。

GVS、DVM和LCM这三个子层共同组成了整个系统的第二层，即Distributed Kernel。

3. 第三层：一次性基础设施虚拟层

这一层可能表述和理解起来有些困难。何谓一次性基础设施？难道基础设施用完了就丢弃？这岂不是浪费么？是的，就是因为所有资源都是虚拟的，所以才可以浪费。Applogic的所谓一次性基础设施，其实是说虚拟机资源和网络资源可以按需求创建，每个应用程序都可以为其分配一个独立的基础设施平台，包括Firewall/Gateway（基于Iptable）、负载均衡器、Web服务器、应用服务器、数据库服务器、日志收集服务器、NAS存储器等，而每个角色都是一个虚拟机。

这些虚拟的基础设施组合起来成为某个应用程序的容器，当彻底删除一个应用程序的时候，容器也随之被删除，这就是所谓“一次性基础设施”的意义所在。这些角色被这一层虚拟成一个个的对象，在图形界面中使用鼠标拖曳就可以创建，同时用户还可以指定这台虚拟机的In和Out网络接口和个数，比如，一台负载均衡器，就需要至少一个In接口和若干个Out接口。同时还需要指定Volume的容量和数量，以及CPU、内存等资源的上下限。被装配好的基础设施可以被复制、粘贴、导出和导入等。

如图20-8所示，用户创建了一台Web服务器、一台MySQL服务器和两台NAS存储器。Web服务器的前端in接口提供用户访问，后端的db接口用来连接数据库服务器，fs接口通过NFS协议访问一台名为content的

NAS存储器，log接口使用CIFS协议访问一台NAS存储器用来存取日志。用户将会在这个硬件平台上运行某种应用程序。

图20-8 创建Web、NAS和DB服务器

图20-9所示则是一个入方向的Gateway，一台负载均衡器后挂4台Web服务器，最后共享一台数据库服务器的基础设施架构。

图20-9 Gateway创建

4. 第四层：网络控制层和应用程序管理层

这是整个Applogic体系的最后一层，也是最后的展示层，所有的底层逻辑无非都是为了实现这最终的展示层，也是Applogic应用程序运行平台交付方式的最终体现。

Applogic以Application为单位向用户交付，在一个网格（也可以说成是集群，包括主机和存储以及网络）上运行多个用户的多个应用程序，比如，Exchange、CRM等，每个应用程序会被分配一个一次性基础设施，每个一次性基础设施又包括了多个角色，比如Gateway/Firewall、负载均衡器、Web和数据库服务器、NAS存储器等，每个角色就是一个虚拟机。每个Application创建好之后，就是一个Package，可以独立操作，与底层硬件无关，用户可以将它带走，在另外一个Applogic网格内导入，便立即可用。

5. 配置实例

图20-10所示为整个网格系统的Dashboard监控界面。

图20-10 Dashboard控制台

图20-11所示为网格中所有已创建的Application的列表。

图20-11 已经创建的Application列表

下面我们通过创建一个带有Gateway防火墙、一个负载均衡器、两台Web服务器、一台数据库服务器的简单的Infrastructure来体验一下Applogic最终交付给用户的接口。

(1) 先从左边的Gateway类别中拖曳一个简单的入方向的Gateway/Firewall。然后在其上右击，从弹出的快捷菜单中选择Property Value命令，如图20-12所示。在出现的窗口中可以定义这台Gateway的一些属性，如图20-13所示。

图20-12 创建Gateway

图20-13 设置这台Gateway的属性

(2) 加入一个8 Out口的负载均衡器。然后右击并选择Resources命令来配置这台均衡器所使用的资源，如图20-14所示。可以配置CPU、内存、网络带宽这三种资源，其接口是由Citrix Xen提供的，这个界面只是调用这些接口来控制Citrix部署虚拟机，如图20-15所示。

图20-14 添加负载均衡设备

图20-15 设定虚拟机参数

(3) 拖入两台Web服务器，并在这个Application的主界面中单击Manage Volume命令来创建这个Application所需要的存储空间。此时界面会调用底层的集群卷管理层来执行部署任务，如图20-16所示。

图20-16 设定存储空间

如图20-17所示，这个列表列出了分配给当前Application的所有Volume，整个网格中的Volume都是按照Application相互隔离的，不同的Application只能看到自己的Volume。

图20-17 当前的所有可用Volume

单击Add按钮添加一块存储空间。名称、大小、文件系统格式，如图20-18所示。

图20-18 添加新存储空间

（4）将Volume分配给需要使用存储空间的服务器，比如Web服务器，在Web服务器图标上右击并选择User Volume命令，如图20-19所示。

图20-19 分配这个新空间

里面默认有一项content卷，我们将上文定义好的code卷映射给这台Web服务器的content卷，如图20-20所示。

图20-20 分配新空间（1）

（5）拖入一个数据库服务器，将上文定义的data卷映射给它的data卷，如图20-21所示。

图20-21 分配新空间（2）

建好之后的最终拓扑如图20-22所示。

图20-22 创建和部署好之后的拓扑示意图

防火墙、负载均衡器和Web Server以及数据库服务器之间互连时使用的IP是用户不用关心的，系统会自动分配，一切力求简化、快捷。唯一需要配置的是整体Application的IP地址等信息。

可以针对每台服务器进行各种属性的配置，如图20-23～图20-25所示。

图20-23 配置虚拟机上业务层的各种属性
(1)

图20-24 配置虚拟机上业务层的各种属性
(2)

图20-25 配置虚拟机上业务层的各种属性 (3)

我们还可以登录到这台服务器上，如图20-26～图20-28所示。

图20-26 登录到对应的虚拟机上执行任务 (1)

图20-27 登录到对应的虚拟机上执行任务 (2)

图20-28 登录到对应的虚拟机上执行任务 (3)

此外，还可以监控服务器的各种状态信息，如图20-29所示。

图20-29 对整个系统进行性能监控

(6) 最后一步就是登录到相应的服务器，安装相应的软件，启动应用软件。

Applogic带来的革命在于它把复杂的底层硬件变得非常简单，通过拖曳对象的方式来装配自己的Infrastructure，并最终以一个适合于某种Application运行的整体服务器+存储+网络环境来交付给用户，佐以底层丰富的附加功能比如Snapshot、Migrate等，为用户提供了一个专业而

且方便的程序运行硬件平台，让用户彻底脱离了苦海。

20.5.2 IBM Blue Cloud

IBM的Blue Cloud，是IBM推出的云解决方案。其基本上是由一堆软硬件有机堆叠起来的。硬件当然包括IBM自己的X系列PC服务器，以及刀片服务器，更少不了P系列小型机，大型机就算了吧，已经是被时代所淘汰的东西了。软件则必然包括VMware Vsphere以及Citrix Xen虚拟机平台了。有了这两个还不行，必须还有一套用于资源自动调度分配管理监控的平台，这就是IBM RDP（Request Driven Provisioning）、IBM TPM（Tivoli Provisioning Manager）、IBM TM（Tivoli Monitor）。有了这些还不行，还需要对应的中间件平台，比如Websphere，还要有开发部署平台比如Rational，再加上其他一些上层的软件套件比如Lotus、Information Manager等。总之IBN借助这个篮云，大大地整合了它所有的软硬件产品，促进了所有这些产品的销售。

IBM宣称蓝云可以在5分钟内部署一台x86平台的虚拟机，P系列小机的部署则需要30分钟左右。x86当然是靠VMware以及Citrix Xen作为平台；而P系列自身就提供了硬件级别的虚拟分区功能，而且也支持分区迁移等高级功能，其灵活程度毕竟不如x86开放平台，所以部署所需的时间也要远远长于前者。

我们还是来看一下这个蓝云是怎么对外呈现的吧。首先见图20-30，为用户登录到云服务申请界面时的入口。用户使用自己的账户和密码登录蓝云资源申请界面。登录成功之后，便会出现如图20-31所示的窗口，其中会显示出这个用户之前曾经申请过的所有资源以及对应的简要情况。

图20-30 登录到云服务申请界面

图20-31 此用户所申请的所有资源列表

在图20-32所示的窗口中单击右下角的Request New Project按钮来申请新的资源，出现如图20-32所示的窗口。其中可以选择用户要使用对应资源的期限，比如本例中选择了呃SEP，也就是9月份的10/11/12这三天，也就是要用三天的时间。

图20-32 指定使用期限

单击next按钮，出现如图20-33所示的窗口，这里可以给本次所申请的资源起一个名字以用来区分，并且可以添加描述。另外，窗口上方可以看到目前云中现存可供使用的所有资源（当然，可能系统并没有列出所有资源，可能会根据目前账户的级别来只列出其可享受的等级的资源）。用户可以选择对应的资源，比如X系列PC服务器，或者刀片服务器，或者P小机服务器，旁边还列出了可用的内存与CPU频率资源，以及对应的存储资源。选择对应的资源然后单击Add按钮，出现如图20-34所示的窗口。

图20-33 命名和描述所申请的资源

图20-34 添加资源

本例中选择了使用刀片服务器资源。在这个窗口中可以指定使用刀片片的数量，AvailableResources显示2，这并不一定就代表此时系统中只剩下两片刀片了。前文说过云中可能都是一些虚拟的资源，这里的2可能就代表此时系统只能再为你创建两台虚拟机了，因为你的账户是铜牌账户。可能对于白金账户的话，这个数字会更高。这里选择了1，下面

还可以选择在这台虚拟机上你想让它安装部署什么操作系统，这里选择了Linux。是否需要部署中间件？如果勾选的话，那么系统会自动将WebSphere中间件部署到虚拟机中。这些定制化的选项，其后端必定对应这样一种情况，比如系统预先将各种可组合的情况独立生成一份虚拟磁盘镜像文件，比如装好WebSphere的Linux系统，或者一份裸Linux系统，或者部署了WebSphere的AIX系统镜像。当用户选择了对应的组合之后，系统后台就将这些文件复制一份或者直接做一个快照来供对应的虚拟机运行，如图20-35所示。

图20-35 已经创建好的资源

如图20-36所示，参数已经选择完毕的资源会被列在下方的窗口中。此时可以继续添加资源，如果添加完毕，则单击右下角的next按钮，出现如图20-36所示的窗口。

图20-36 系统后台开始部署对应的资源

此时系统后台就开始根据用户指定的参数来部署对应的资源了。比如在Creating Project过程中，系统后台可能会在云管理平台中创建对应的条目和记录等，这一步是比较快的；Reserving Infrastructure过程中，系统后台会调用对应的虚拟化管理平台比如Citrix Xen来部署对应的虚拟机，比如将预先定制好的虚拟磁盘镜像文件复制一份到对应的存储空间，然后创建新虚拟机，然后挂起磁盘，启动。整个过程完毕之后，单击OK按钮结束整个过程，出现如图20-37所示的窗口。

图20-37 刚才创建的新资源已经列出

单击这个新创建的资源进入资源详细信息页面，如图20-38所示。可以看到系统已经自动部署了对应的操作系统、中间件，以及配置了IP

地址。

图20-38 资源详细信息页面

用户此时就可以使用Telnet、SSH或者远程桌面等来连接这台服务器从而执行后续的定制化部署了，比如安装其他软件等。此时这台虚拟机就完全是你的了，愿意怎么搞就怎么搞，搞崩溃了也没问题，只需要让运营商给你回滚一下就可以了，用什么回滚呢？当然是快照技术了，或者在虚拟机层实现，或者直接在底层存储中实现。当然，快照能回滚到最近的时间点粒度，也是一种服务级别。比如你购买了高粒度的快照，那么或许就可以回滚到5分钟之前的状态；而你如果只购买了低级别的快照粒度，那么你可能就只能回滚到一天前的状态了。还有各种各样其他类型的服务级别，这些都需要用户在购买云服务的时候与运营商签订SLA（Service Level Agreement），SLA中会规定所有详细的服务条款。

其他运营商的方案与3teraApplogic以及IBM Blue Cloud如出一辙，不管是IaaS、PaaS或者是SaaS，其不同的只是交付方式的区别。读者可以从本案例中窥见一斑。

20.6 乘风破浪——困难还是非常多的

云这个东西并非都是优点。作为一个刚刚兴起来的概念，其机遇与挑战并存。

20.6.1 云的优点

1. 为企业解除了投资浪费的后顾之忧

买了服务器买存储，买了存储买交换机，买完了硬件买网管软件、买运行维护服务，磨合期结束时，发现需要扩容需要购买新的硬件，耗费大量的电能却不是每时每刻都在计算、都在存储。这种问题在迁移到云之后将会不复存在，整体资源极大地得到节约。

2. 绿色节能，社会成本降低

随着云服务的发展，网民只需一条高带宽的网络连接和一台处理能力很弱但是耗电很低的瘦终端机和显示器即可享受各种信息服务，包括3D渲染服务，甚至高性能显卡也无须再购买。节能不但只针对于使用云服务的用户，运营商自身通过使用虚拟化整合的弹性数据中心，也可以实现节能增效。

3. 廉价

随着网络带宽逐步增加和云的发展，云服务将会变得越来越廉价从而走入千家万户。届时人们会发现与其购买一台高性能PC主机和大容量存储硬盘、高性能显卡，倒不如购买同样感观的云服务来得方便和便

宜。云服务时代，一切你本地所能享受到的IT服务，都可以在网线那头得到满足。

4. 角色转变

云服务将企业内部的IT基础架构从围绕应用系统开展的模式全面转向以服务为宗旨开展的模式。没有云之前，企业的IT基础架构一切工作都要围绕应用系统来开展，基础架构的配置水平严格由应用系统决定，应用系统以及使用应用系统的部门成为资源的掌控者；而私有云形成之后，IT基础架构部门变为资源的掌管者，其他部门都变为消费者，向云来要服务。这种政治角色的转变，是比较微妙的。

20.6.2 云目前存在的问题

1. 稳定性和安全性的问题

换个角度，在运维方面，你觉得谁更专业？你还是云？如果你觉得你更专业，那是另外一回事；如果你觉得云更专业，你认为将同样故障率的Infrastructure放到你这和放到云里哪个更保险？数据都是你的，只不过换个人保管，既然你认为云更专业，那就应该相信云。

这和银行是一样的道理，你总是认为银行有更保险的措施来保障你的金钱，唯一区别是钱都是一样的，而数据具有唯一性，所能做的就是多几份冗余。说到数据所有权的问题，比较有意思的是，Facebook前段时间修改了其服务声明，宣称它拥有数据的永久的所有权，并且有权力使用保存或者发表在该网站上的任何内容。笔者想这只是特例，一切都是可以谈的。个人认为云系统中基础设施和数据本身的安全性问题大可不必担心。

另外，共有云所面临的一个最大问题就是如何彻底隔绝两个用户的存储资源，用户总是有这种担心，比如互为竞争对手的A与B公司共同选择了云运营商C的服务，A与B各自的数据如何才能确保底层被完全隔离，因为用户往往认为只有这样才能够真正做到自己的数据不被对手通过某些后门黑客手段所窃取。

2. 涉密问题

另外一个担心就是将数据存储在不信任的地方，如何解决涉密问题。这确实是一个原则问题。这方面依然是可以通过技术层面解决的，比如将数据源放到本地并在服务器端采用加密方式，用户控制密钥等。

3. 平台迁移问题

怎样在不影响已经运行的业务的前提下将其迁移到云平台，这也是一个比较头疼的问题，需要提供商给出最佳的解决方案。

4. 骑虎难下的问题

一旦用户选择了某家云服务提供商，如果之后想更换提供商，则几乎是一件无法找到好办法的事情，因为你的业务、数据都已经部署在这家云提供商处了。更换提供商就意味着将数据业务进行迁移，并且伴随着业务停机。想要做到停机时间最短的迁移，就需要两个运营商之间形成合作关系，这不是一件容易的事情。

5. 不兼容问题

不同云服务商的架构不同，所提供的访问接口也不同，随着云的发

展，势必会有云之间通信的要求，就像多个网络之间互联形成Internet一样，云也一样，这方面需要一个像IP地址和TCP一样的标准出现。目前Amazon的AWS（Amazon Web Service）系列接口已经逐渐在成为标准，这些标准包括：SimpleDB：simple DB接口（与Hadoop中的bigtable，hypertable之类比较像）；RDS：亚马逊的Relational Database Service接口；SQS，即Simple Queue Service服务（类似message queue这种服务）；CloudWatch，监控整个云；EBS，即Elastic Block Stor接口，用来控制块数据访问；IAM，即Identity and Access Management接口，用来控制和认证用户行为等。

6. 带宽问题

网络带宽始终是Web 2.0和云时代的首要瓶颈，网络带宽不飞跃，服务也不能飞跃。

7. 运营方式

云要想盈利，重点是运营方式。用什么样的方式来获得用户的关注，是需要不断努力做的一件事情。

总之，适用于企业级业务的云，目前还尚未得到普及，随着技术和时代的发展，企业级云服务最终将会在市场上占有一席之地。

20.7 千年之梦——云今后的发展

20.7.1 云本质思考

存储资源+计算资源+软件资源，最后虚拟为IT服务资源，也就是最终形态的Software as a Service (SAAS)。抽象来看，这不就是一场虚拟计算机么？硬盘+服务器+操作系统和应用软件+显示器。其实目前的IT技术都没逃出这个模型，都是这个模型的各种变形体。

说明： 比如Browser—Webserver—Application Server—DB Server，就相当于显示器—显示卡—主机—硬盘。再抽象一下，就是显示器—硬盘，人们最终需要的是结果并将其显示出来看到眼里听到耳朵里，看完了还得存起来或者从硬盘上取出来。人们需要的就是显示器和存储设备，中间的任何过程，从最终用户的角度来说，是根本不去关心的。而云服务的出现，恰恰就是这种思想的一个呈现和回归。

既然用户最终需要的不是服务器也不是一堆一堆的硬盘也不是机房也不是整天吹冷风的空调，他们只不过想要自己的数据被展现出来，看到生产的结果并且储存起来备用，就这样嘛。好吧，你不喜欢机房，也不喜欢看到一堆一堆的机柜，你不愿意看到你花很多钱购买的服务器，在系统不繁忙的时候，CPU利用率几乎等于0，磁盘IO利用率几乎等于0。即便这样，你愣是无奈地乖乖的让这些家伙持续耗费着电，磨损，磁盘依然高速旋转着就为了时不时地接受那无关痛痒的几次IO操作；系统管理员在巡检完后，无所事事地聊天玩游戏而你却得按月支付工资，你同样也不愿意再为了这些你认为已经让你觉得浪费很大的投资而再次投资购买什么网管软件或者监控软件之类的东西，同样你更不愿意看到

花钱购买的硬件在几年后被完全淘汰并且不被厂家支持了；你也不愿意每月缴纳那高昂的电费和雇佣系统管理员的开销。你每天望洋兴叹，昨天服务器Down机了，生产停顿30分钟，损失10000元，今天硬盘损坏，换了一块硬盘花了2500元，谁知道明天又会有什么麻烦事呢？

你好，我们这里有机房有机器有存储有网络，有非常专业的操作人员和维护人员，我们帮你运行应用程序，我们帮你存储，你不用担心硬件淘汰的问题，淘汰了算我们的。我们给你一个IP地址，你用浏览器登录这个地址打开网页就能发送指令并且看到你的结果了，而且还能看到你的结果被妥善保存了，这样你满意否？满意的话，咱们谈一谈你需要付多少钱来购买我们这种服务吧。你很痛快地付了款，我们很痛快地将你眼前的乱摊子转移到了我们这里，你花了钱，我们给你收拾了乱摊子。从此你可以解脱了，你可以专心去做更重要的事情了！

这就是云的本质的一种，云其实就是一种服务，云本身不是一个物质，它不是你能摸到看到的，但是它要基于物质才能显现它的功效，你可以把它看作某种能量的体现，即存在于物质之上的东西，正因为这样，才让人晕，不细细体会是看不清这层形而上的东西的。《易经》有云：“形而上者谓之道，形而下者谓之器。”所以，下器者，谓之服务器+存储+部署管理软件；上道者，谓之“云”。所以，云是一种道，是一种方式和方法，而并不是某个设备，某个软件，当然云这种道要用软件来实现，而且是模块众多的一套复杂软件，同时要用硬件来承载。

所以，云与速度和性能没有直接关系。就像前几年IPv6刚被炒作的时候，铺天盖地的媒体乱鼓吹说IPv6时代下载一部高清晰电影只需要几分钟，笔者当时听了就一头雾水，IPv6协议上再优化，也不可能超越其底层所承载的物质的极限，即互联网络的硬件基础，链路不提速，何来上层的提速？云也是一样，云本身并不一定就是一个高速高冗余性的东

西，而是说其底层的硬件一般使用并行计算集群和存储集群，在这个基础上云方能表现出更大的效能，况且云也不是为了提速而生的。云的最重要的作用其实是廉价高效地利用资源并且将硬件拥有成本和管理成本降低为几乎等于0，成本模式全部转为服务费。当然，性能也是云的一个重要考虑。

其实云早就存在了，在互联网诞生的那一天，云就一直存在，慢慢发展直到近两年才被炒作起来。云其实早已落地生根，只不过现在才钻地面并且惹人关注了。可以这么说，互联网服务就是云服务（注意，不代表云服务就一定是互联网服务），所以有人提出了EAAS，Everything as a Service。IT服务即云，即云服务。

目前存在一个误区，认为云必须是集群，必须是让你不知道数据在哪里，或者数据必须在所谓“很远的地方”却让你感觉到其实很近，才算云。其实这只是云发展过程中的其中一个形态而已。一台Web服务器在互联网上提供服务，你所知道的只是它的域名和IP地址，你同样也不知道它是一台单一域名和IP的单主机，还是用一个IP对应多个域名的虚拟主机，你更不知道网页数据存放在哪里，它可以存放在本地硬盘，SAN存储设备，或者是通过长距离网络从一个“很远的地方”取回数据而发送给你，难道这不是云么？同样是云。这恰好就回归了云的本质，即用户根本不用关心服务方采用什么方式，哪怕它就是一台简单的服务器。

而且上文中也论述过，云是“上道者”，而集群是“下器者”，下器者是在不断上升和发展的，而上道者则是轮回往复的。但是绝不能说云与底层硬件架构无关，硬件架构越庞大性能越强，则云层也要跟着适应其发展，会产生更多的具体细节功能。云系统硬件正在向超大规模集群、高可靠性可用性、高动态扩展性方面发展；云系统的软件即硬件之上的云层，正在向功能多样化、管理简便化、按需分配资源、廉价方面发

展。

在Web 1.0时代，网络尚未广泛普及，带宽极低，56K Modem上网时代，使得服务者和浏览者双方都受到限制，使用者不可能通过这么低的带宽来观看在线视频或者在线音乐，更不可能下载或者上传一些容量很大的资源。这直接制约了服务方的发展，造成互联网服务形式过于单一，大多是缺少互动的简单媒体形式比如简单的网页、静态的新闻之类。

而随着网络硬件技术和软件技术的发展及人民生活水平的提高，在几年的时间内，1Mb甚至更高带宽的网络终端走入了千家万户，直接突破了底层限制，而且计费方式从按时间收费转为包月包年。这直接使得互联网成为一个随时可访问的、永远在线的永久媒介。网民们纷纷从传统媒体转到互联网，而这又直接催生了各种互联网服务，比如音视频、聊天、博客、网络硬盘、P2P、高清电影等。带宽的增加和不限时在线，让网民们毫无顾忌地上传视频、音频等，恨不得塞满服务器，一时间使得数据暴增，互联网一下子步入了Web 2.0时代。Web 2.0时代就是一个网民互动、数据暴增、服务多样的互联网时代，互联网的作用真正的得以体现。

在这个时代，越来越多的人选择将自己的数据存放在服务方而不是自己的本地硬盘，因为自己的硬盘永远不够用，而且也懒于备份。人们通过网络硬盘、邮箱、公共FTP、网盘等方式将自己的数据上传并且得到永久保存，更重要的是，在哪里都可以访问到自己的数据，只要有网络。服务提供者可以在此基础上实现分级的服务，比如免费服务和VIP服务等多样化的服务方式。这便是云的方式，云的思想。可以说Web 2.0时代将是云服务的温床。最近又有人在炒作Web 3.0的概念，加入物联网元素来翻炒，这里就不赘述了。

20.7.2 身边的各种云服务

网站、QQ、MSN、ICQ、Camfrog、网络硬盘、网络音视频、网络账本、网络日记、在线文档阅读、在线游戏等，仔细想来，这些就在我们身边的东西，都是云服务。我们通过浏览器来接收Web网页和通过Web网页提交自己的请求却不关心服务方的具体架构；我们通过聊天工具和其他人就像在同一房间内一样通过文字语音视频聊天而并不知道自己和他人是否连接了同一台物理服务器；我们上传自己的资料到网络硬盘却不关心服务方有没有及时备份或者我的数据是不是被服务方擅自迁移到了慢速的SATA硬盘存放；我们在网页上阅读各式各样的文档却根本不用安装对应的文档浏览工具。

还有，中文输入法的革新，输入法已经是与时俱进，充分利用互联网的便捷。比如某拼音输入法，可以随时更新专项词库，各行各业的不断出现的新名词统一由服务器收集并且提供下发更新，你再也不用一个字一个字地敲，而且其同样步入了云服务时代，每个用户可以将自己的词库信息上传到服务器，这样，不管你在哪台电脑上使用输入法，词库信息都会实时同步以满足你的输入习惯和要求。并且服务器整词输入时，程序还会去服务器端动态检索是否有相同音的词被其他人输入过，也即是所谓“云输入法”。不过这个云却是非常实用的，比如某些同音词，单靠软件智能是无法正确输入的，但是借助云，你可以看到其他人对于这个拼音所输入的词汇，你把所有人的知识拿来用了，这确实非常让人兴奋！同时你曾经输入过的词汇也会动态地被发送到服务器端保存，也会贡献给他人。这样就形成了一个“个体—整体”的系统，个体的行为贡献给整体，整体又反过来影响个体、加速个体的发展，个体整体之间相互促进，形成一个有机的智能整体！

20.7.3 进化还是退化

独立计算回归集中计算，是退化还是进化？都不是，是轮回。公共的集中分时计算早在40年前就出现了，随后逐渐萎缩。世纪相交时刻的前后几年，用户都各自购买Infrastructure自行计算和存储，但是为何进入新世纪之初，又逐渐兴起回归到集中计算时代呢？笔者认为这依然应了中国的一句古话：“分久必合，合久必分。”即万物皆在一个轮回中不断地发展，到一定程度就会改变成最初的形态，但是承载它的物质是连续不断地积累和提升的，所被轮回的只是其上的那层能量，谓之道！硬件不断地发展，其上的道却是不断轮回地往前发展，这种现象存在于万物之间，不能用什么科学解释，只能说这是自然的规律。

比如存储系统的演变，从独立存储DAS发展到网络集中存储SAN，你也可以称SAN为一个云服务，很恰当，而如今随着盘片密度的不断提高和SSD固态芯片存储的发展，好像存储系统有那么点回归独立存储的趋势。比如在1U的服务器上插满12块2.5英寸的SAS硬盘，其提供的容量和速度也是相当可观的。

理想： 就像笔者个人曾经预测的一样，如果存储介质真的能够发展到惊人的高密度和高速度，谁还会插个卡再用线缆连接到某个很远的地方只为了存储数据呢？更甚一步，集中存储轮回至独立存储之后，什么时候可能再轮回到集中存储呢？无线时代。随着无线电技术的发展，人们发现可以用无线电通信直接到地球上甚至宇宙中某处来存储数据，此时我还需要在本地使用大容量的芯片么？至此我更加坚信“上道者”的轮回往复。然而，“下器者”是否一定也是永远前进发展而不轮回呢？笔者给不出答案，如果古玛雅人的预测是真的，那或许是的，只不过一个轮回要经历太长时间而已。

20.7.4 云发展展望

网线可以代步，网线比宇宙飞船速度快，网线上可以传来香喷喷的菜肴和服装，网线可以做任何事情，给我一根网线，你可以关我禁闭，如果要加上一个期限的话，可以是一万年。随着互联网和电子商务的发展，越来越多的人成为宅人，他们的工作以Homeoffice为主，生活上也是足不出户。这些宅人们充分享受云的便利，腾云驾雾，通过网络订餐送水购物，活的怡然自得。正所谓：白云巅上白云斋，白云斋下白云仙；白云仙人布网线，原来仙人也是宅。

当入户网络带宽达到很高的数值之后，更高级别的云服务将彻底颠覆PC市场，用户都转向了集中计算和集中存储。有人或许会质疑，就算是1Gb的以太网到户，其提供的带宽果真能够满足将来的存储需要么？

这一点我们可以做一个论证，现在你就可以试试看，从移动硬盘上复制一个文件到你的本地硬盘，速度有多少？充其量四五十兆吧，这算好的了，1Gb的理论速率是100MB每秒，除去各种开销，咱们就给他算30MB每秒。目前来讲，这个速率相当于本地硬盘的大部分操作情况下的速度了，这就完全可以满足要求了。

或许还有人质疑，入户带宽达到1Gb的时候，那时候的应用程序和对数据的需求量一定是得寸进尺，影片不仅是1080P了，而是4080P，家里都用高清电视墙或者投影仪了。那也不用担心，就算是4080P，1Gb的带宽也足够流媒体的在线播放要求。

我们要理解的一点是，云即是服务，我们通过网络拿过来的是内容的表示，而不是内容的裸存储或者计算。比如原来我需从网络上下载

一个数据库程序安装文件，好几GB，我还是需要等上一段时间才能下载完，还得安装配置，不方便。

但是你忘了一件事情，云服务时代，是不需要你自己运行数据库的，你只需要从云里拿到最终处理并且表示的数据即可。这些处理后的内容有多少呢？很少的网络流量，除非是高清视频等Rich Media。或者你需要下载一个Windows 7的安装镜像，10GB用来安装在你本地，“无法安装Windows 7，程序未能检测到本地硬盘”，你将会得到这个提示，你还是忘了一件事情，你此时还需要操作系统么？你在操作系统中得到的服务，云都提供了。或者又有人质疑，“谁说的？操作系统能安装SCSI卡驱动程序，云能安装么？”如果真有人这么问，我只能无语，除非这人真是一个患有“驱动程序安装”强迫症患者。或者还有人质疑，云能玩网游么？能玩对硬件要求非常高的大型3D游戏么？当然能了，而且现在就已经实现了。

说明： Crysis《孤岛危机》是我向往玩一玩的一个画面效果至今无人能敌的游戏，然而它对显卡的要求实在变态，至今尚未进行体验。那么，对于一款连高端显卡都无能为力的游戏，云有何能耐应付？你忘了一件事，家用显卡无能为力不代表专业渲染集群也无能为力。好莱坞用专业渲染主机集群来实现Crysis的渲染实在是小儿科了，渲染之后的高清图片帧通过网络传输至用户的显示器，用户的游戏操作数据通过网络发送至集群游戏渲染服务从而实现互动，整体体验与在本机运行游戏一样。有人甚至用手机玩Crysis，这不是做梦，已经真的实现了，这个云渲染系统正处于测试阶段，相信不久便会真正对外开放。

云服务是否也会步ERP或者 ITIL的后尘呢？ITIL尚未落地，因为它

根本就没有落地的根基。而云是个早已落地的产品，目前正在以各种形式成长。要说云是革命的话，那这句话要在互联网发展初期就该说。

云就像一个早已被播种，经历了几十年之后长大成人并且一下子出现在人们面前的风华正茂的少年！云服务相对于ERP和ITIL来说有本质的区别，虽然三者都是对于信息化实现的方式和方法。云有其存在的根基，因为其作用在各个层次——基础设施层IAAS，平台层PAAS，内容计算表示供给层SAAS，任何人都有机会并且已经在享用云服务，凡是与互联网有关系的人或企业都离不开云服务了。

而ERP和ITIL属于一种企业生产和管理方式和流程的信息化过程，需要这种服务的企业少之又少，就算有了需求，实现起来更是难上加难，其本质原因还是底层物质难以承载上层的能量。就像你让一个入道尚浅的人去修炼一种高深的秘籍，那是基本不可能的事情，因为其悟性和道行根本就达不到那个境界。然而，ERP和ITIL是否可以借助云服务来让自己站住脚呢？已经有一些基于云服务模式的ITIL或者伪ITIL产品出现。让我们静观其变吧！

云，将会大行其道！

20.7.5 Micro、Mini、Normal、Huge、Gird弹性数据中心

目前已经有厂商开始打包卖弹性核了，或者直接出售或者出租软数据中心。比如将若干刀片与高密度磁盘柜以及微型交换机打包到一个或者几个机柜中，上面再覆盖以弹性层，比如虚拟机管理系统以及分布式存储系统，将这样一个微型的弹性软核心作为一个可提供IaaS服务的整体交付给用户。

或者再在这个软核之上覆盖一层业务展现于管理层，直接交付到

SaaS层。一柜或者数柜交付的弹性基础设施，可以称之为Micro Cloud，有些厂商干脆使用集装箱当做容器来盛放十几个机柜，里面消防、监控、供电、散热等都已经做好，这种集装箱交付方式又可以称其为Mini Cloud。对于以集装箱为交付单位的Mini Cloud，当需要在短时间内迅速部署IT资源，或者临时性的应付一下业务峰值而需要增加IT资源的时候，这种集装箱就可以派上用场。比如发生了战争，某重要数据中心被摧毁，那么便可以使用集装箱快速重新部署；再比如某机构机房空间不足，那么便可以临时性的使用集装箱来应付，或者直接将集装箱作为外辅助数据中心。根据交付层次的不同，可以只交付集装箱+供电冷却消防监控设施+机柜，也可以将一定容量的计算与存储设备整合进去，做到这一层便算是IaaS层了，之后还可以继续向其中整合更上层的东西，比如PaaS、SaaS。再大一个档次，就是正规的数据中心了，数百个机柜的容量，可以称之为Normal Cloud。那么再大一个档次呢？也就是多个数据中心之间的互联网业务互联了，这样就形成了Seed Cloud，每个Normal Cloud都作为Seed Cloud的一枚种子而存在。再之后呢？这些大种子将会弥散开来，种子的粒度越来越小，小到每一个节点都可以作为一个种子，这就是大范围互联网网格时代了，也就是云2.0时代，此时可以称其为Infinite Cloud。整个发展过程和形态如图20-39所示。

图20-39 云发展的5个阶段

20.7.6 弹性层的出现将会让数据中心拥有两套性能指标

弹性数据中心将会有两个主要指标：硬核的计算和存储能力、软核的计算与存储能力。比如一台8刀片+嵌入式交换机+高密度磁盘柜的机箱，其能提供的原生固有操作系统承载数量为8，计算能力为若干Flops或者若干TPC-C指标，原生存储容量为100TB；但是增加了弹性层

之后，其可扩展的OS数量变为 $8 \times n$ ，等效计算能力被灵活扩张；对于存储，内置的Thin与Dedup引擎可以增加存储的弹性，分布式的文件系统或者卷管理系统可以增加存储系统灵活性。这样，这个弹性核可以在计算与存储方面具有两个指标：原生计算与存储资源指标和弹性扩张之后的计算与存储资源指标。比如原生存储容量为50TB，弹性存储容量为100TB甚至500TB，视Thin与Dedup的效能而定。

20.8 尘埃落定——云所体现出来的哲学思想

20.8.1 轮回往复——云的哲学形态

说明： 不知道大家是否玩过游戏《质量效应2》，或者看过《黑客帝国》三部曲。前者是游戏，后者是电影，两者都是科幻题材。但是二者却都体现出了一种轮回效应。前者的最终Boss是由人体和其他无机物共同组成的一个人形的庞然大物；在后者的结局中，成千上万的机器也在空中形成了一个形。虽然这种情景只是人的想象，但是绝不是空穴来风，它体现了一种哲学，一种轮回嵌套的哲学。再比如科幻电影《黑衣人2》的结尾，把宇宙描绘成一个怪兽手中的玩具，其实宇宙是另外一个世界中的玩具球。还有，所谓：“一沙一世界，一花一天堂。”这些其实本质上都体现了这种轮回往复的世界观。

云也是这种世界观的体现。表面上看好像云是一个独立的个体，其实它内部是由无穷的二级个体在起作用，二级个体又可以再继续往下分；同样，向上，云可以合并，多个云可以共同组成一个更大的云，轮回往复。

云除了体现形态上的轮回之外，还体现了个体之间的关系，大量个体的动作自然而然地形成了整体的动作，而这个整体却又反过来体现了大多数个体的共同意愿和思想以及物质形态。比如麦田怪圈、高楼大厦、蜘蛛做网、天空云梯、鱼群、蚁群等。我们可以在空中用一只巨大的手来画出麦田怪圈，或者用橡皮泥来捏出一堆高楼大厦模型，但是我们却无法想象通过个体之间的协作和大量个体的劳作，渺小的个体也能够建造出规则的巨大的建筑，并且还可以通过外形形态来传达各大多数

个体的共同思想。

最后，人群可以体现人的意识的组合；计算机群也一样可以体现机器的意愿。目前来讲，机器体现的其实都是人的意愿，但是不排除机器自身进化之后可以产生自己的意愿。物质群和意识群，一个在下，为阴为器；一个在上，为阳为气。这两者螺旋上升地发展。

云是一种技术思想上的轮回表现，也就是从独立计算转变为集中计算的过程，仿佛又回到了上世纪的大型计算机时代。但是，虽然思想轮回穿越了，承载这种思想的物质可没有轮回，物质总是不断前进的，不会回头，除非世界末日到来。

同时，云系统本身也是由阴和阳组成的。阳表现为云系统中的商业运营模式以及所有的软件；阴则表现为承载这些思想以及运行软件所需的硬件物质基础架构，即服务器、存储和网络等设备。阳又是一种形而上的东西，而阴则是形而下，形而上者为道/气，形而下者为器。整个系统是一个阴阳和合相衡的生态系统。

20.8.2 智慧之云——云的最终境界

云也可以理解为智慧的结晶，一个超级智慧实体，就像《黑客帝国》中的Matrix角色一样。由大量个体组成的智慧体，个体只是按照给自己的那一小部分规则在运行，而整体则显示为具有个体属性的，但又有升华的超级智慧。

最后，人的意识组合成了社会，计算机的意识也会组成计算机社会，云的进化，最后就会成为一个计算机社会。这便是智慧之云，它反映了全体个体的智慧结晶，并且不断地进化。

20.8.3 云在哲学上所具有的性质

(1) 大规模的云由大量个体组成，个体的具体属性可能不同，但是其本质都相同。

(2) 任何个体进入云中将被冲淡，将被吸入成为云的一部分。

(3) 个体连续地工作才形成了整个云的动作。

(4) 个体的损耗、损毁、代谢交替对整个云影响不大。

(5) 云有它的寿命，虽然可以通过代谢来不断更换衰老的个体，但是最终会像生命体一样完全消亡。

(6) 大云可以捕获、吞并和消化其他小云。

(7) 云有智能，云会进化。

(8) 云和云之间将会形成生态关系，形成云社会。

(9) 社会即云，全人类即云，太阳系即云，银河系即云，宇宙即云。宇宙也是个体，多个宇宙再组成云，轮回往复，无穷无尽。

20.8.4 云基础架构的艺术与哲学意境

另外，我发挥了一下想象力，将云想象成为一部精密机械，并画了一张图，如图20-40所示。希望通过这张图能够让大家更加深刻的认识云这个东西。这张图片其实是可以运动的，大家可以联系我以获得这张图片的动态版。这张图是对一片小云种子的一个总结。最中心的物质本源，也就是硬核心。在这层硬核心之外，包裹一层弹性存储层，包括可

灵活扩展的集群SAN、集群NAS或者集群/分布式文件系统等，以及Thin、Dedup等增加数据弹性的技术，还有增加管理灵活性的虚拟化技术。之外再覆盖以弹性计算层，最后覆盖以资源管理层，最外面则是运营层，包括业务展现子层与运营管理子层。至此这片云就彻底运转起来了。

图20-40 小云的5层结构

大家再仔细分析一下这张图，会发现其中包含有无穷奥妙。你先想象它是一台电动机，其中每个圈都可以旋转，谁来给它提供能量呢？当然，云数据中心必须有足够的电力，电动机要旋转，需要有电刷给其供电。大家可以看看业务展现层与运营管理层，这两个模块是不是很像两把电刷，将用户需求这种动力不断地提供给中心部件，从而让这台电动机持久地转动。如果没有了用户需求，或者你已经想不出足够新颖的业务展现方式，那么就算有电，这个数据中心也无法再运营下去了。

- 计算存储硬核层：处于这台精密机械的最核心层次的，就是硬核了，也是运转最快而且最硬最实的一个角色。在数据中心的表现为大量的服务器和存储设备，这个核心是云数据中心的物质本源，也是密度最高的实体。硬核相当于云中的种子。
- 存储弹性软化层：这一层紧密地包裹在物质实体之外，通过分布式、集群、虚拟化、Thin、Dedup、Snapshot、Clone等技术，将原本高密度的硬实体充分软化，为上面的层次提供一个弹性软化的基座。
- 计算弹性软化层：在弹性软化的存储基座之上，计算资源也通过各种主机虚拟化技术手段被充分地软化。
- 资源管理调度层：物质硬核+富有弹性的软化计算与存储资源层，共同构成了一个系统内的阴与阳相合的资源核心。资源管理

与调度层起到一个适配内外层次的作用，外层阳气的生发需要底层物质源泉的积聚、分配和运化，如何将这个系统内的资源进行良好的调动与配合，便是这一层的任务。这一层中存在诸多角色，比如资源监控、资源分配与调度、并行计算分配与调度等。

- 业务展现运营层：这一层则是阳气外发外散与运化升华的至极之层次，也是整个云数据中心的精神本源。中心的硬核为太阴，这一层则为太阳，为最终将辐射展现出去的一层，也是外界直接可见的一层。表里相合，一为实一为虚，一为阴一为阳，一为物质一为精神。

整个云数据中心便是一个阴阳表里虚实相合的一个有机体。这也正像一个星云，不断孕育出[微软用户1]星体，对应着云基础架构之上不断孕育着各种应用业务服务。生物体、社会、企业、公司、宇宙，其实都是这样演化的，那么可以这么说，系统即云，云即系统，云是大统一的系统模型。

另外，大家可以继续领悟一下，这片小云种子，是否就是一台计算机呢？硬核心就相当于硬件，看得见摸得着，外层的弹性存储层，相当于计算机的存储系统，任何计算机启动时首先都要去从存储系统里读入代码执行；再外层就是操作系统内核层，基于存储层的支撑，内核得以启动；再往外就是操作系统管理界面层了，利用界面来管理和分配各种计算机资源，其中并行计算调度模块也相当于操作系统中的线程调度器等角色；最外面一层，也就相当于计算机的应用程序层了，各式各样的应用程序，对应着云中各式各样的业务展现。各种应用（业务展现）可以在一台计算机上（云中）迅速的装卸载（部署）。那么，“云即计算机”，这句话，不记得之前哪位提到过，至此我也彻底理解了。云为何就是计算机？云由大量计算机组成，而其堆叠之后的样子和架构，仍

然还是一台计算机。计算机各处总是体现着轮回的形态。为何呢？因为它骨子里就是由计算机组成的，它永远造不出异形，只会造出它自己，除非它有自己的强烈向往，希望自身向某个异形发展。这种行为，骨子里已经根植到了基因当中，这种上下联系看上去非常微妙。

对比一下：

（1）计算机加电，硬件启动→云数据中心硬件核心层启动。

（2）从磁盘读取代码以便启动OS→云中的数据承载层。各种分布式FS分布式DB，key-value DB等。

（3）OS启动→云中的虚拟计算层，生成大量VM。

（4）启动到用户界面→云中的管理层，比如微软System Center，Novell Cloud Manager，思科UCS的Unified Manager等。

（5）内核的线程调度器→云中的并行计算调度层，比如Mapreduce 以及其衍生物。

（6）OS提供的开发API，VC，Java→云中的PaaS开发平台。

（7）各种运行在OS上的应用程序→SaaS展现层，各种云业务。

又比如各种分布式文件系统，其本质是什么呢？其实还是本地文件系统思想的外散。本地文件系统通过一个跟入口，然后一级指针、二级指针、三级指针，一直到最后一层0级块用来存放最终的文件数据。而分布式文件系统，大家思考一下，对称式分布式文件系统，其各级元数据其实也都是分部到所有节点当中的，比如一级指针用1个节点承担，二级指针用2个节点承担，3级指针用3个节点承担，这样可以将文件系

统分布到非常大的范围。或者干脆把所有指针元数据放到一个节点中，那么这个节点也就是MDS，这样就属于非对称式集群文件系统了。但是不管怎么弄，其本质其实就是本地文件系统的思想。

至此，“轮回”的道理在计算机世界已经充分展现了出来。

20.8.5 纵观存储发展时代——云发展预测

如图20-41所示为存储系统的发展时代。这张图所要表达的两个最重要的地方，一个是所谓“小云时代”与“大云时代”，另一个则是“分”与“合”的轮回变化。

图20-41 存储系统架构变迁发展

早期的直连存储，每个计算节点都有各自的存储系统，计算节点之间的存储系统没有共享，属于分；今天的网络存储时代，所有计算节点可以共享访问同一个存储系统，属于合。当前，由于单台网络存储系统性能受限，集群存储系统的地位开始上升，利用大量分开的存储节点来获得更高的性能扩展性，以满足数据爆炸时代的需求，这又属于分；而在将来，将是一个业务爆炸的时代，各种各样的互联网服务层出不穷，如何快速满足业务部署需求？传统数据中心，部署一台新服务器和业务，需要至少一天的时间，比如上架、安装OS、安装支撑系统、安装应用系统、测试，而如果加上前期采购等流程，则需要更长时间。而云的出现，利用弹性核心，将底层大量分开的计算或者存储节点屏蔽，抽象为各种IT服务，使得部署一个新应用可能只需要几个小时的时间就可以了，云整合了所有IT资源，这又属于合。而在遥远的未来，随着技术革命的到来，计算与存储将会发生质的变化，单个节点的计算与存储能力发生革命，另外，互联网带宽的革命，也会促使大范围的网格节点互

联，共同组成一个整体，而不是向某个固定的云中获取资源，这又属于分。

小云也就是不久的将来会大行其道的、由小规模的核心作为种子所生成的云[微软用户2]。而大云则是借助高速互联网而组成的大范围的网格型的核心，大家共同参与组成云。此时，任何节点可以申请加入这个云而成为云中的一粒种子，不断贡献着资源而同时也吸收着资源。其实P2P就相当于一个大网格云，只不过网格云时代P2P的不仅是电影，而是一切IT资源了，比如存储和计算。假如你有一个视频需要渲染，但是你的机器自身的显卡太差，那么你可以直接利用P2P渲染平台服务将你的任务提交，其后台就会将这个任务分解到所有拥有高渲染能力的节点中去并行计算，然后结果汇总到你这里，这个速度将会是相当快的。

在网格云时代，此时整个系统又属于分了。在这个大云时代，互联网上的所有节点都会有高接入带宽，比如万兆以太网到户，再加上IPv6，使得每个人都变成了互联网运营商，那么之前被大集中到运营商以及各个数据中心中的资源，在这个时代将会重新流入到互联网所有节点中。之前被大集中在运营商数据中心的数据业务，此时会被大范围分散到互联网的各个节点中去，此时的形态又属于分了。如果说集群存储时代对应着数据爆炸，那么小云时代便对应了业务爆炸，数据与业务共同在几个点上爆炸之后，必将经过弥散的时代，这个时代也就是大网格云时代。在这个时代，整个互联网内的资源被虚拟地整合起来，人们可以利用P2P技术运行超大程序，渲染图片以及海量存储等其他各种服务。那时候的运营商，有一种很重要的角色，这个角色就相当于分布式文件系统上的MDS，它会给整个互联网提供一个用于P2P共享的分布式平台，保存所有互联网元数据。比如大分布式文件系统、大并行计算调度系统，都会有运营商来掌握和部署，将整个互联网范围内的资源整合

了起来，数据和资源弥散在整个互联网范围。

另外，小云自身也相当于大云中的一粒种子，目前各地都在兴建小规模云数据中心，这些建好的数据中心，将来会成为种子，然后各自联合起来形成大范围云。我们姑且称小云时代为云1.0时代，或者也可以叫它种云时代。而大云时代为云2.0时代。云2.0时代的时候，之前云1.0时代的种子便完成了它的使命，成功地发芽开花并结果。

那么在大云时代之后，按照之前一路分分合合的变化规律，势必又会变为合的形态。这里先这样推测，然后再找论据，它凭什么要合呢？你说合它就合么？显然不是的。我们畅想一下，那时候计算与存储将会发生革命，比如从电路计算与电路存储，转为量子计算或者生物计算与分子存储。大家知道，生物细胞中的DNA可以存储大量的信息，可能一个细胞可以存储的数据堪比一块硬盘，那么一片树叶中将可以存储海量的数据。那么谁来读取其上的数据呢？当然是靠核糖体，核糖体就相当于磁盘中的磁头。细胞也可以用来计算，一个细胞中包含多种模块，每一个都被精确设计，来完成一系列的生化过程，这个过程如果可以被翻译为其他逻辑，那么就可以被用作高速计算。那个时候，将会出现利用革新技术生产的计算机，整个时代又会回到最原始的形态，也就是大量终端通过网络连接到中心计算机来获取资源。

纵观云发展史，我们再往高层思考。一开始，PC在LAN内访问服务器，这种状态可以映射到现在的Micro Cloud；后来，IT组织越来越大，数据中心出现，组织内的PC可以跨越多个LAN被路由到数据中心LAN内访问所有资源，此时可以映射为Mini或者Normal Cloud；后来Internet发展，使得多个Internet上的服务器可以相互通信相互要资源，PC可以连接到Internet访问其他LAN内的服务器，这种状态可以映射到大范围的Seed Cloud；那么再后来，Internet上的资源被一定程度地弥

散，当然数据中心等云种子依然存在，只不过资源被极大地弥散到了所有节点之上，此时便映射到了Infinite Cloud，也就是大范围Gird了。

请看一下图20-42所示的IT系统基础架构一开始都是孤岛，烟囱。后来存储开始被集中起来。再后来，存储的上层，也就是计算层，也被集中整合起来。再后来，统一了开发平台，其上孕育出各种应用。这也是当前的新型IT基础架构的形态，也就是人们常说的“云”基础架构。存储从主机脱离出来成为SAN，那么整个IT系统基础架构的共享，就成为了Cloud。可以看到一层层的向上侵蚀，但是APP一层自身能被融合起来么？恐怕不能了，每个业务都要对应一个APP。那么这个系统正在干什么呢？它下一步将会是什么形态呢？

图20-42 IT基础架构演进示意图

IT系统最终是不是要长成这个样子（见图20-43）全部融合。一个操作系统实例可以管理所有服务器、存储、网络资源，也即所谓“数据中心操作系统”或者“分布式操作系统”。这个DOS（Datacenter/Distributed OS，不仅寒了一下，确实是个轮回，到了DOS时代了）可以有两种模式，就像分布式文件系统一样，一种模式是把多个文件系统实例用目录的方式虚拟，而另一种模式是直接在底层虚拟。

图20-43 分布式统一操作系统

那么DOS也可以有这两种类似的方式，要么就用一种简单松耦合的管理角色，比如微软的System Center、Novell的Cloud Manager等；要么，也就是要引出的另一种彻底底层融合方式，一个操作系统实例，高速网络的出现，将总线延伸到计算机外部（请翻回阅读本书第2章，外

部网络就是计算机总线的延伸），将内存的RDMA，这些都变得可能，此时多个物理上分离的节点可以直接进行总线级的通信和内存直接寻址，那么一个分布式操作系统也就可以在此基础上诞生了。

这个操作系统将会把整个数据中心中的所有元素统一管理起来，包括底层的供电、消防、监控等，及上层的存储、计算、网络资源管理和调度，以及各模块间的消息转发中间件平台，乃至最上层的应用业务展现开发平台。所有这些元素将被一种像微软Windows操作系统一样的统一平台来操作与管理。这又是一个大轮回了。同时，存储早就准备好这一天的革命了，各种分布式虚拟化存储已经成熟。此时，整个系统确实从骨子里表现为一台大计算机了。这个DOS会将任务拆分为颗粒分配到底层所有计算节点中去，这也正如传统OS将每个线程分配到CPU的每个核心中去执行一样。

这种操作系统如果真的出现，其早期必然不能够满足高实时性计算的需求，只能作为一个庞然大物级别的系统来处理一些非高实时性但是却拥有极高的总吞吐量的系统，对于高实时性需求依然需要独立的系统。其实这种演进就相当于目前的传统操作系统与嵌入式高效实时的精简操作系统的区别了，谁也离不了谁，总要有一种通吃一切的庞大系统，比如Windows、Linux等，也必然要有一些尖兵操作系统比如VxWorks等。随着规模的庞大，整体颗粒度必然变大，微观运动必然相对变慢，但是总吞吐量数量级也必然升高，这就像一个细胞的运动与地球自转的比较一样。

那么，这个系统究竟是否会朝着这个方向演进呢？让我们拭目以待。

20.9 结束语

当云的底层架构从数据中心集中的Infrastructure变为全Internet上的节点之后，也就是网格云计算，这种庞大的网格在节点间相互交流后会逐渐进化，或者不由自主地表现出一种全体无意识性的结果，这种结果往往是惊人的，就像蚁群一样，会创造出惊人的结果。再比如，眼前的例子：人肉搜索。

现在提出云是否会太超前？我们知道，客户端并行访问数据的概念早在1994年就被Gibson提出了，当时被设计为一种对象存储系统。这么优越的设计为何最近才被纳入业界标准呢？各个分布式文件系统厂商相继推出配套的并行访问客户端。究其原因还是因为硬件、网络的发展。x86服务器硬件已经相当廉价，1GbE以太网环境比比皆是，10GbE马上也要普及开来，而软件相对变为了瓶颈，传统只通过单一链路访问目标的做法已经用不满网络带宽了，为此pNFS等标准被制定出来。

我们再反观目前的云，云确实是比较超前的东西，但是看看现在的互联网带宽，不用说1GbE，10MbE都还没有普及。在这种低带宽的网络环境下，互联网用户暂时只能享受到一些不需要高带宽的服务，比如在线文档阅读等，而这些服务将其归类到互联网服务比较贴切，尚未达到云服务的层次。而当互联网带宽革命之后，云此时才会被大规模释放出来，用户用一个瘦终端即可享受到任何IT服务，连操作系统都不需要安装。

最后，用图20-44表达一个观点，即一切皆是在业务驱动下，不管是计算、传输还是存储。没有业务驱动，计算、传输与存储便没有了任何意义。业务又是人类需求的体现。

想象一张图，在世界地理版图上，分布着若干个大云团，周围佐以一些零散的小云团，整体形成一个云网络，形成一个计算的脉络。这是何等壮观之景象（如图20-45所示）！不知道至此你是否还会看云而晕呢？云计算/云存储/云服务，是否真能成为IT发展的天堂？让我们拭目以待！

图20-44 业务驱动下的计算、传输与存储

图20-45 世界云版图

附录1 存储系统问与答精华集锦

本问与答精华集锦全部收录自作者几年来在各大存储技术论坛以及个人博客中针对广大网友提出的问题的回复内容。作者对这些内容做了总结和归纳，作成了问与答的形式。其中有对于存储技术基础问题的问答，也有针对本书内容的问答，还有对一些方案设计等问题的问答。作者希望读者通过阅读这些问答内容，能够解决自己心中的一些疑问，从而对整个存储系统的理解更深一步。

1. 关于“文件”与“块”

请教个问题：什么是文件数据，什么是块数据？它们有什么区别？

块泛指底层磁盘上的扇区组合，某个文件可以对应一个或者多个这样的块。客户端访问存储系统的接口有两种：网络文件系统接口和SCSI或者ATA接口。前者指的是CIFS和NFS；后者可以是SCSI、FCP、iSCSI、ATA/IDE、SATA、eSATA、1394、USB。前者在网线上的数据三元素是文件名、起始偏移量、读写字节数，后者在线缆上传递的三元素是起始扇区号码、操作码（读、写等）、要操作的扇区数量。

文件系统的作用就是将块虚拟成文件。如果磁盘阵列中集成了自己的文件系统，则可以虚拟这些块，向外面提供NAS接口访问，此时盘阵的存储空间在主机端表现为一个挂载上的目录（不需要格式化，直接向其中读写文件，也就是NAS访问方式）。如果盘阵自己没有文件系统虚拟化功能，则只能依靠客户端，也就是主机端上的文件系统来管理此盘阵向这个主机映射的磁盘。

关于“数据块”级和“文件”级之间的差异，能不能给我讲透彻一点？

你好，Block级别就是直接通过读写存储空间（磁盘，LUN，Volume）中的一个或者一段地址的扇区来存取数据，文件级别则是通过读写某个文件中的一段数据。比如你是存储设备，我是主机，我说：请你将LUN1上的0~127这128个扇区的数据给我，你给我了，咱俩之间就是Block级的访问。如果我这样和你要数据：请你将X目录下的

tellme.txt文件的前128字节传给我，这就是文件级别的访问。前者俗称SAN Block访问，后者俗称NAS访问。

2. 何谓“NAS”？

想知道**NAS**的功能具体是怎么实现的？可不可以举例下？谢谢。

你好，**NAS**就是，前端用网络文件系统提供服务，后端用任何方式的存储空间。可以说，只要前端通过网络文件系统服务的存储设备就是**NAS**。

3. 关于**FC Chip**带宽、接口带宽、链路带宽

我在网上看了一下，比较有名的是**Sansky**博客上提出的理论带宽计算是根据每控制器上的主机**FC**芯片和磁盘**FC**芯片计算，**Sansky**指出以**DS4800**为例：**DS4800**虽然主机端口和磁盘端口都是8个**4Gb**接口，但实际上每个控制器内部只有两个**4Gb**主机 **FC chip**和两个**4Gb**磁盘 **FC chip**，两个控制器加起来是4个**4Gb FC chip**，所以两个控制器的最大传输速率是**4×4Gb**。我有点疑问，如果我们来看**IBM DS4700**两款产品的话，**IBM**提供的带宽大概是**1270MB/s**和**1550MB/s**，这两款产品都是双控制器产品，每控制器芯片分别为1个和两个，磁盘芯片均为1个。如果按照如上计算，是达不到**IBM**宣称的技术参数的。老冬能否帮忙解答一下？谢谢！

你好，确实达不到的。那个参数具有误导性。确认一款产品的带宽，考虑6个地方即可：前端接口数，前端芯片数，后端接口数，后端芯片数，前后端之间的总线带宽，内存带宽。任何一个地方受限制，就会达不到表面的标称值。有时候多个端口是共享一个**Chip**（芯片），那

多半就只能共享带宽，但是也不排除有使用高端线速交换的芯片，就和交换机一样，虽然看起来口不少，可是背板带宽小的话，那就不能保证所有口同时线速交换。加上前后端之间总线不够的话，那就也有瓶颈。同样，如果内存带宽不匹配，也会有瓶颈，因为磁盘数据都是先入内存再入前端口。

你说的高端线速交换芯片是怎么回事？是否就是说只要背板带宽足够大，最终磁阵的吞吐量就能由前端和后端接口的数目决定？

所谓线速，就是指所有端口都以自己的额定速度同时交换，而所有端口都是连接到背板上的，其中需要核心芯片管理，而核心到这些所有端口之间的总线，就需要支撑这么大带宽的同时交换。盘阵吞吐量的决定因素，这个就更复杂了，与IO、控制器、通路带宽和效率、后端磁盘类型和数量均有关。如果不考虑这么复杂的话，吞吐量可以近似认为由前后端Chip和接口数目决定。

老冬，我刚才问了一下我们的研发人员，是不是**Sansky**的说法有些不对，带宽不能根据**FC**芯片计算？我们的研发人员说，**4G FC**芯片每秒处理能力不止**4G**，只要在总线带宽和**CPU**的处理能力能包含的范围之内的话，就可以根据主机接口数来计算理论带宽。

不懂你具体的意思。**4G FC**芯片处理能力不止**4G**，这个说法不太明白。**FC Chip**上对于接口的编码速度部分是严格精密的，包括**Timer**和**Serdes**串化解串化器，是严格的**4Gb**，至于不止**4Gb**的说法，我不清楚，可以让研发人员直接留言。**FC Chip**可以管理多个口，如果其所管理的多个口是共享的**FC Chip**的核心**FC**处理逻辑，则不管多少口共享**4Gb**带宽，如果每个口各保持独立，即每个**FC**处理核心处理速度可同时保证多个接口同时并发地线速以**4Gb**速率独享带宽运行，则这样的话，吞吐

量与接口数量可以成正比。所谓的4G FC Chip处理能力不止4G的说法可能从此而来。

4. iSCSI与Windows下动态磁盘的问题

前两天在现场看到一奇怪问题，**iSCSI**存储直连**Windows 2003**主机，每次重启的时候磁盘管理下存储的那块硬盘都变为脱机状态，右键重新激活后就能用了，分区也还在，发现是动态磁盘的简单卷。网上搜了一下，有哥们说**Windows**的**Initiator**不支持动态磁盘，给微软的工程师电话咨询，他当时也不能确定，好在没什么重要数据，得到客户许可后将磁盘设置为基本磁盘后就正常了。哪位大侠碰到过这种情况吗，能不能给个解释？

MS的**iSCSI Initiator**用户手册中明确指出了不支持动态磁盘，原因就是网络服务加载晚于动态卷服务。**iSCSI Service**加载太晚以至于动态磁盘管理系统认为丢盘了。**Linux**下的解决办法是编辑加载设备的配置文件表明本设备属于**netdev**。不过MS应该着手解决这个问题的。

5. FCAL环路性能问题

大多数磁盘阵列都是通过两个控制器后端的端口，组成**1/2/4**条**FC-AL**环，来连接所有磁盘。**FC-AL**仲裁环的协议规定，同一时刻只有两个设备能传送数据，也就是说，在一条**FC-AL**环里面，控制器的一个后端端口充当了发起者的角色，环上的一个硬盘充当了目的地的角色，在一个时刻里，后端端口发出数据读写指令，只有一个硬盘能响应这个指令并传输数据。那是否就意味着，一条**FC-AL**环的总体性能，取决于一个硬盘的读写性能？**1500**转速的硬盘，持续读写带宽不到**70MB/s**，**IOPS**不到**400**。那像**IBM DS 4800**、**EMC CX-80**之类的，

总共**4**条环，后端的性能岂不是只有**280MB/s**、**1600 IOPS**？除了以阵列里**Cache**来提高速度，我不知道有没有别的办法，或者，磁盘阵列内部采用的不是工业标准的**FC-AL**？

这个问题非常好，而且也非常经典。要解释这个问题，需要明白三点：

- FCAL的传输通道的确同一时刻只允许两点间独占通道带宽来传输数据。
- 控制器在有足够IO请求的情况下绝对不会让通道闲着，会充分利用带宽。
- 磁盘的外部传输率和内部传输率。FCAL环路上存在多个设备的时候，由于控制器的轮询策略充分利用带宽，整个系统在外体现为一个永远都在读写数据状态而不是寻道状态的大虚拟设备（本书有描述），一旦某个设备需要寻道，那么就让其他设备来传输数据来弥补所浪费的时隙，所以整体系统可以发挥出一个单一设备的内部传输率。

下面是详细总结。

当FCAL环路上存在多个设备的时候，控制器向某设备发起IO之后，该设备需要一定的寻道时间，而此期间内AL环处于被释放的状态，此时控制器依然可以向另外的设备发起IO，也就类似于先把该做的命令全部下发，待某个设备寻道完成请求将数据返回给控制器的时候，往往是多个设备都处于积攒状态，也就是它们都干完活了，准备交差了，而此时只能排队一个一个来，大家都鼓着劲呢。明白了这一点，我们就往下看。

关于IOPS数值的矛盾：

IOPS与吞吐量是一对矛盾关系。在关注IOPS的环境下，IO Size往往比较小，因为只有较小的Size才不至于充满带宽达到瓶颈，所以，要达到较高的IOPS，IO Size需要比较小。而这种情况下控制器将IO请求发送给设备之后，多个设备处于积攒状态，它们会仲裁从而一个一个地分别得到传输数据的机会。由于IO Size很小，所以每次传输数据很快就结束，这样，一个IO就飞快地完成了，而上一个设备的IO完成之后，下一个设备接着也会很快完成，因为它已经处于积攒状态，待返回的数据早已在Cache中准备发送。这样的话，这个整体系统对外就表现为一个永远在完成IO而不需要寻道的虚拟设备。而如果AL环上除了控制器之外只有一个设备，那么环路就必须等待它寻道，因为寻道的时隙内，AL环上已经没有任何其他设备可工作了。然而，AL环的这种弥补寻道时隙的效果也不是设备越多就越好，不同的设计和产品都有自己不同的最佳设备数量，目前的经验值为64个，也就是环路总容量的一半，超过这个值，性能不会再有提高，甚至有所下降。我们可以推论出另外一个结论，也就是，慢速设备，比如寻道时间长的设备，越是慢速设备，组成AL环路之后其带来的整体提升越大；越是快速设备，高规格的设备，组成AL环路之后，提升的性能越有限。这就是AL环或者其他共享总线/环方式弥补设备自身处理产生的时隙的效果。

关于吞吐量/带宽值的矛盾：

经过上面的描述，我们已经对共享总线/环传输方式的底层机制以及其效用有了一个很好的理解了。在重视和追求高吞吐量，也就是充分利用带宽的环境下，IO Size往往非常大，以至于在较低的IOPS值下就已经可以吃满通道带宽了。往往这种情况下，上层所发出的IO都是连续大块IO，以至于AL环之上的设备寻道时间可以大大降低，这就使得设

备更快地处于积攒态，准备向外发送数据。我们知道磁盘外部传输率由于磁盘内部不断换道被打断，致使其数值较内部传输率降低了大约20倍。而AL环的效用就是弥补寻道所浪费的时隙，所以整体系统的外部吞吐量就被提升了上来，从而解决了这个矛盾。

6. 关于RAID层面的IO并发与Stripe Depth的一系列问题

看书产生的疑问，书中讲IO并发几率的时候说，“对于RAID 0，在两块硬盘的情况下，条带深度比较大的时候并发两个IO的几率为1/2”，我理解为条带深度比较大的时候容易产生并发。但是在书中分析“分析以上过程”这段里说“在某种特定条件下要提升性能，让一个IO尽量扩散到多块物理硬盘上，要减小条带深度”，我理解为扩散到多块硬盘才能并发IO，要减小条带深度。但是，这两种说法是否矛盾了？可能是新手，理解不正确。

前者是并发IO，后者不是并发IO，后者描述的情况是大块连续IO的情况下，数据适当分散有利于提高传输速度，以便饱和链路带宽。要IO并发，还是要单个IO速度最大化？自己决定。

你指的大块连续IO，意思是，数据是同时分散在Disk 1、Disk2、Disk3上写，还是先写Disk1，完了再写Disk2。那如果是同时写的，那么它和并发IO有什么区别？这部分看得有些疑惑，希望指点。

问得很好。IO是并发的，如果分散的话，所有盘同时读写，当然，如果用FCAL，微观还是指令顺序被发出，但是发出之后，磁盘干活还得需要时间。这个间隙中，控制器就可以轮流发送指令给其他硬盘，脑子转一下，想想这个模型，转起来之后，理想情况下，整个RAID组就相当于一个磁头永远在读写数据而不被寻道所打断的大虚拟磁盘，这个

书中也描绘了。这样，外部传输率就能达到一块磁盘的内部传输率了，当然，理论理想情况下。

在**RAID 0**中并发为什么条带深度越大越容易并发**IO**？并发**IO**跟条带深度有什么关系？这点始终理解有困难。书**81**页的表中 **IO Size/StripeSize** 这个表达式是比值大小的意思吗？呵呵，书认真读了，但是理解有点困难，我想把前面的章节弄清楚点再读后面的会比较好。希望老大以后多指点。

这点怪我，在原文段落中没有提前再次声明一个**IO**到底是什么形式。一个**IO**的三大件：操作类型、起始扇区地址、长度。其次，**RAID**组上的虚拟磁盘的**LBA**地址是局部竖向（**Segment**内）、全局横向（**Stripe**内）排列的，单盘的**LBA**是绝对竖向排列的（当然此处不考虑盘片柱面等物理因素）。这样，条带深度（**Segment**）越大，就可以同时接受多个**IO**，越小，**IO**中的**LBA**长度就越容易跨满这个**Stripe**。

又来提问题了，呵呵。书中**83**页提到**RAID1**的并发**IO**不支持随机**IO**关于这句话——“写**IO**因为要同时向每块磁盘写入备份数据，所以不能并发**IO**，也不能分块并行”不太理解。

呵呵，每天来一问，每天有提高！

一句话：每个磁盘同一时刻只服务于一个**IO**，**NCQ**中的**Out of Order**技术另当别论。这样你的问题就明白了，思考一下。

有一些小疑问。我们平常在配置**RAID**的时候，**Stripping Size**是预先设置好的，比方说**64KB**，然后我的**Segment Size**是等于**Stripping Size**除以它所组成的数据盘，而且冬瓜头也说到了**Segment Size**大小应该是扇区的整数倍，那如果说我的数据盘是**7**块，那我的**Segment Size**

应该怎么算啊？ $64/7$ 除不尽啊？这一点没有看明白，老大能帮我解释下么？谢谢！

好问题。首先，**Stripe Size**是包含**Parity**盘在内的。关于除不尽的问题，不一定。看具体实现了，可能输入64，则自动变为48或者96，都有可能。总之，看具体实现。这一点我已经向某国外OEM厂商的研发人员咨询过，得到了肯定的答复。现在一些机器都只给出设置**Segment Size**的接口，而且估计有些机器所谓的**Stripe Size**其实就是**Segment Size**。

书中提到：**IO Size /Stripe Size** 较大，顺序IO，连续读，提升了 N 系数倍。

而随后又说：系数=**IO Size /Stripe Size**...，大于等于1。这里的系数为什么大于等于1？应该是小于等于1吧。

这里大于1小于1都可以，因为 N 可以随机。

《大话存储》P82，读，**IO Size /Stripe Size** 较小，并发IO，随机IO，提升了（1+并发系数）倍，这个1是怎么来的？

这里1就是单位1，起码要比1块磁盘高，所以需要1+系数。

请问冬瓜头，当**IO Size**小于条带深度的时候，**RAID 0**写数据是不是只写往一个磁盘？为什么很多地方又强调条带化时写数据是同时往所有的磁盘而不是写完一个磁盘的**Segment**再继续写另一个磁盘的？如果是这样的话即使**IO Size**小于条带深度也应该是每个磁盘都写入一部分数据吧？看着有点糊涂，请指教。

当**IO Size**小于**Segment Size**的时候，这个IO一定是只落在一个磁盘

上了。同时向多个磁盘写某个IO Data的情况是IO Size大于Segment Size的时候。

那是不是可以这么理解：写数据是以**Segment**为单位的，比如有5个硬盘做**RAID 0**，**Segment**大小为**1KB**，当一个**IO Size**为**2KB**时，它只将数据并行写入其中的两个磁盘，而其他3个磁盘对这个**IO**是不起作用的？如果是，数据是依据什么寻址，即决定写入哪两个磁盘的？另外，存不存在一个**IO Size**不是**Segment**整数倍的情况，此时数据是怎么写入的？还有，是不是所有**RAID**条带化都是基于这个原则？请指教。

那是不是可以这么理解：写数据是以**Segment**为单位的，比如有5个硬盘做**RAID0**，**Segment**大小为**1KB**，当一个**IO Size**为**2KB**时，它只将数据并行写入其中的两个磁盘，而其他3个磁盘对这个**IO**是不起作用的？（是的）。如果是，数据是依据什么寻址，即决定写入哪两个磁盘的？（数据逻辑地址和物理地址**Block**的映射关系是由控制器做以记录的，控制器会依据映射表来做翻译，书中讲了，请仔细阅读。）另外，存不存在一个**IO Size**不是**Segment**整数倍的情况，此时数据是怎么写入的？（存在，该怎么写就怎么写，**IO**最小单位是**LBA**扇区，也就是**512B**，一般情况下。近年来吆喝着要增大**LBA**容量到**4KB**，尚未成型。记住，**Segment**只是一个逻辑概念而已，磁盘不理解**Segment**，控制器对磁盘发起的实际**IO**都是以一段连续的**LBA**为单位。）还有，是不是所有**RAID**条带化都是基于这个原则？（基本原理类似，但是具体实现看各厂家设计。）

7. 关于“**Block**”的意义

再请教一个问题，**RAID**中的**Data Block**和我们平时做**RAID**的时

候选择的**Block Size**是否是指同一参数？

平时做RAID的时候的“Block Size”，要看RAID控制器厂家如何定义的，不同厂家叫法不同，具体咨询厂家。

8. 用Iometer测试时所选择的参数

我用Iometer 测用DELL/EMC CX 600 磁盘组建的RAID 5 的IOPS。我的做法如下：

我在Windows XP 上运行IOMETER，在 Linux 上运行dynamo，Linux与磁盘阵列之间用FC连接。我要测试最大IOPS，所以我照着指导书上说的调整成512B；100%读；0%Random。问题：（1）这样设置对吗？磁盘阵列的扇区大小应该也是512B 吧，读比写快，顺序读也比随机要快，我觉得应该对吧。（2）我没有设置QD，所以我测最大是120000左右，EMC官方说它的IOPS最大为150000左右，我是在一个电脑上运行一个worker试过、2个worker、3个workers。（3）如果设置QD，对于我的配置，应该怎样设置？（4）最后一个问题是最郁闷的：我就那一天测试出来是120000左右，隔了一天测试，今天测试，最高才只有50000左右。我不知道为什么。

120000相对于150000来说，已经不错了。官方的承诺值在实际测试或者使用的时候是达不到的，官方都是王婆卖瓜自卖自夸的。

QueueDepth可以设置大一些，不过与底层的一些因素也有关系，FC卡就是一个主要瓶颈处，有的FC卡写死了QueueDepth，不管上面多大，到了它这都得排着。多个worker也是好办法，增加并发度，总之120000应该是这台设备的最大IOPS了。关于你问的QueueDepth怎么设置，自己去找，不同设备不同设置法。IO吞吐量突然降低，可以排除下列几个

因素：测试工具应用端设置出问题，FC卡驱动或者Firmware进入不稳态需要Power Cycle，配置变更，外部存储设备处理瓶颈（通过盘阵监控工具进行监控），网络传输上发生问题（查看本地接口及交换机状态和配置）。

9. RAID 5在一块盘故障之后如何继续服务IO的问题

如果一个**RAID 5**在创建时没有指定热备份盘，那它在一个磁盘失效之后，能够继续响应用户的读请求是很自然的，那它能否继续响应用户的写请求呢？它的逻辑地址空间是怎样的？发到失效设备上的写请求如何处理？

RAID 5坏一块盘依旧可以写，坏的如果是Parity盘，数据依然完整写入数据盘。如果坏的是数据盘，则一个写需要牵一发而动全身了，此时如果某IO只更新坏Segment的数据，则读改写Parity计算公式中的“老数据”一项变为未知，所以只能读出所有其他数据盘的数据进行校验然后将新校验值写入校验盘即可。这个问题不难理解，请再仔细用脑思考。

10. 关于磁盘扩展柜上的IN和OUT端口的問題

接触存储时间不长，问个小白问题，请入门的兄弟给解答下。控制器后面的主机端口上面有**IN** 和 **OUT**之分，麻烦给解释下有什么区别。如果将控制器主机端口（**IN**）与**MDC**相连可以映射到阵列中的相关**LUN**，如果与（**OUT**）相连则看不到磁盘，这是为什么？

完全看设计了，其实in和out都是连接在一个Port Bypass Circuit芯片上的。按理说连到out上也可以进入FC Loop，但是有些设计可能要求必

须接到in一条，然后再接到out一条，就可以同时认到双路径。以上说的都是Host直连这个in和out。out常规情况下是用来级联更多扩展柜的。

11. 关于磁盘上的“剩余空间”的问题

ext3是众所周知的日志文件系统，在写一个文件到磁盘的空闲区域时，采用先写文件数据，再记录元数据到日志，最后写元数据到磁盘的步骤，这样可以保证数据的一致性。但是如果文件系统下发的写请求是对非空闲的磁盘块（以前已经有内容）进行更新呢？前面所述的步骤就不能保证一致性了，这种情况下**ext3**是怎么处理的呢？

前提，所谓“非空闲”和“空闲”，是针对FS来说的，磁盘自身没有空闲或者非空闲。FS既然决定了要覆盖某个文件的某个块，就证明原来的数据已经不需要了，此时覆盖也无妨。FS是通过bitmap来记录磁盘上哪个扇区为空闲，哪个正被文件占用的。如果覆盖到一半，MetaData未写入，此时依然是一致的，因为bitmap尚未更新，标记的这个扇区依然是空闲的，虽然扇区中实际数据变化了一半。如果楼主理解的“一致”是业务层的一致，那就是不一致，文件没有被真正写入，此时上层应该报错提示程序写入不成功，或者不报错，看需求。但是此时FS是一致的，也就是FS不需要fsck。

12. 关于实际磁盘阵列产品使用FC Class Service的问题

FC协议文档提到：**FC**仅提供了错误检测手段，但大多数情况下**FC**只是将检测到的异常上报给**ULP**，是否重传、重传哪些数据由**ULP**决定。分**Class**级别的，有的**Class**就提供重传，有的则不提供。确实是这样，文档中提到，由于**SCSI**提供了重传机制，因此**SCSI over FC**时，一般采用**Class 3**（无**ACK**）。但文中用括号写了这么一句：**In**

actual SCSI operation, class 3 is used for simplicity of implementation. 其中**simplicity**让人生疑：**Class 3**只是一种简化方案，真正运营时会采用**Class 3**吗？

看情况了，据我所知某阵列使用的确实就是**Class3**，而且我推断其他阵列几乎也都会使用**Class 3**，因为**SCSI**已经有了机制，没有必要外壳层再多此一举了。

13. 关于**Cache Mirror**测试用例的问题

双控，支持**Cache Mirror**。请教，如何设计测试**Cache Mirror**功能的测试用例？还有一个问题：存储控制器的**Cache**一般是单独的**Memory**还是从整个系统的**Memory**中分配一部分来做**Cache**？

设置控制器向磁盘**Flush**的频率到最低，然后测试机发起突发大量IO，期间人为Down掉目标控制器，然后检查磁盘上是否成功写入了数据，成功，则有效，不成功，则**Mirror**出现问题。这里还有个问题，取决于控制器的设计，如果某个IO恰好断在控制器尚未返回**ack**给测试机，则另一个控制器是否可以继续保留状态而返回**ack**？如果不保留，则程序应当报错，检查此IO的上一个IO是否成功，成功，**Mirror**有效，不成功，有问题。

谢谢冬瓜头！有两个问题：（1）存储控制器为何没有调整**Flush**频率的选项？（2）测试机写数据的时候应该复制多大的文件为好？用**IOmeter**这样的工具写数据不太合适吧？

IOmeter写了什么你不清楚，如何检查？得自己写测试程序。最好是写小东西，比如每次一个LBA，其中包含容易识别的二进制。

即使自己写了程序，怎么验证硬盘中的文件是通过**Cache**写进去的，还是通过**Cache Mirror**写进去的？

同一时刻发送大量IO，控制器缓存不会立刻写到磁盘，此时Down掉这个控制器，另一个会接着写，过一段时间就去读出来验证。你再仔细想想**Mirror**的基本原理。

哦，另一个接着写是关键，这样写完后，过一段时间读出来验证，如果文件不完整，就说明**mirror**有问题。我这边主要是没有考虑要结合多路径来测试，如果没有配置**LUN**多路径，一个控制器Down掉了，读写就停止了。另外，因为有你说的“恰好断在一个**IO**还没有返回**ack**之前”，所以需要文件复制测试可能有问题，而需要自己写**LBA**的小工具，对吧？

其实写文件也不是不可以，只是可控性不好。写入大量小文件，Down机瞬间到底写了哪一些，你不知道。

可以写一个大文件吧，比如说压缩的**RAR**文件，如果**Mirror**出了问题，那么在读出来验证的时候，解压**RAR**文件肯定会有问题；但是如果读出来的**RAR**文件解压没问题，那是否能说明**Mirror**一定没有问题呢？

写大文件很有可能触发控制器的连续写盘动作，就测不出来了。

Carlhewei（一哥卡尔）说道：使用“小文件”有更大的机会去测试，大文件的话**Cache Destage**可能会太快而做不成有效测试。“小文件”可能不是个好名词，正确的应该是：自己写一个工具，对象是裸设备，对指定的起始**LBA**写入**Data Pattern**，比如从**LBA 1~2**之间写A，**LBA 2~4**之间写B。有了工具后，就是选址写**Data Pattern**。选址尽量拉远并且随

机，每次写一块。你的工具可以把发出的LBA、Data Pattern和大小记录到log里。然后用冬瓜的方法，可以Hard Reset Controller（就是像断电一样的Shutdown，但是不用拔出来），然后根据log 把数据读出来对比。

14. 冷热插拔与Rebuild

请教下各位，**RAID 5/0+1**是否支持在**Normal**状态下，切断电源，更换**Disk**（**RAID 5 1disk**、**RAID0+1 less than 1 pair**），然后**power up**，这两种阵列是否支持这样的冷插拔的**recover/resyncing**？恢复机制是怎样的？

冷插拔和热插拔一样，起来之后便会rebuild。

15. 如何实现RAID的动态扩容？

（1）**TFTP**能否支持跨网段的传输？（**PS**：今天测试了好半天，最后发现貌似不行，但是也没有搜到相关的官方说法。）

（2）假设一个**RAID 01**由四块**100GB**的磁盘组成，用一个**150GB**的大容量盘更换其中的一块磁盘，等重建完成；然后更换第二块磁盘，再等重建完成.....依次更换四块磁盘容量为**150GB**的新盘。这样第四块盘在开始重建过程中会不会和前面的三块盘没有条带化的**50GB**组成条带（因为现在四个盘都已经是**150GB**了）？简而言之，如何实现**RAID**的动态扩容？

TFTP基于**TCPIP**，除非这个协议必须用到广播，否则怎么不可能不支持跨网段，只要IP能Ping通，**TFTP Server**起来，就可以。（好像有些说法认为某些**TFTP**软件使用**ARP**广播，致使不能跨广播域。）

第二个问题完全看RAID控制器的实现方式了，传统上50GB是浪费了。但是不排除有些控制器的算法不同，能利用起这50GB。由于是一块一块地替换的，所以多出来的区域的条带可能是不连续的，不管怎么说，具体算法，还是设计者说了算。如果是校验型RAID，想利用起多余的空间，就需要将最大的盘作为校验盘以便容纳随后加入的盘，如果加入更大的，则最大的盘顶替当前校验盘，这样就可以继续容纳随后加入的盘了。

在这里多说两句。RAID组扩容后的数据分布有两种处理方式：一是Redistribute，二是Append。前者，在新盘加入RAID组之后，控制器立即将组中原来磁盘尾部的数据读出来写入新加入的盘，重算Parity，平衡数据分布，所以叫Redistribute，很耗费资源。后者，新盘加入RAID组中之后，控制器不做任何额外的数据迁移操作，而只是在逻辑和物理地址对应表中作变更，相当于在RAID组中插入了一个一个的Segment碎片，逻辑上，这块新加的盘的空间被追加的逻辑地址的尾部，而物理上却是琐碎的不连续物理空间。随着前端IO向这块逻辑尾部空间的持续写入，控制器会根据地址映射表将新数据写到对应的物理空间中，也就是前者牺牲了一时的性能来保证地址依然是规则连续映射的，从而保障了后期性能；而后者则是在潜移默化中对后期的性能造成了一定的影响。

16. 流媒体服务器使用何种存储协议的问题

应用：流媒体服务，流媒体处理模块和存储介质物理上分开，前者在单板A，后者在单板B，二者通过以太网总线连接，高带宽（大约在10~20Gbps），且需要保证传输可靠性。问题：A、B之间应该采用何种存储协议？

我的想法如下。

方案一：iSCSI

优点：标准主流协议，可扩展性好，对以太网没有特殊需求。

缺点：**TCP/IP**协议开销较大，为保证高带宽，需要**TOE**技术，成本较高。

方案二：FCoE

优点：不甚了解。

缺点：**FCoE**与**CEE**均未完全标准化，即使标准化，对以太网有额外需求（需支持**CEE**？）。

方案三：SCSI over Ethernet

优点：简单？

缺点：小众标准，不知成熟度如何。

SCSI over Ethernet，目前有产品么？

个人理解，**FCoE**的成本主要集中在：

（1）**FC**协议处理芯片（能否用**CPU**完成？）；

（2）普通千兆或万兆以太网卡。

用哪种协议，真不好说，得实测才行。**File**协议和**Block**各有优缺点。不过本质受限于底层链路，差不太多。目前**FCOE**还不知道是不是

昙花一现的东西。另外如果单纯为了FCOE而FCOE的话，没有意义。如果原来系统中有FC的SAN，现在想融合到以太网中而且有迫切需求，那可以考虑。不过，目前FCOE都是使用专门的适配卡和专门的交换机，与传统以太网根本不可互操作，目前实际主要应用于大型运营商，准备同时投入高速以太网与存储网络的，可以在适配器、布线、交换机方面成本有所节省，离行业市场客户还差很远。16口8Gbps FC交换机也就两万多元一台，一台同样口数的10Gbps的FCOE交换机，贵了去了。但是对于运营商那种大型模块化交换机来讲，FCOE交换机由于融合了两种网络，反而比单独为每种网络都配一台交换机要便宜了。

17. FC是否可以使用主机CPU模拟的问题

想外接**FC SAN**，主机是否必须配置**FC HBA**卡，还是可以用主机**CPU**模拟？

若能模拟，**CPU**开销是否很大？

模拟不了。起码要有张FC卡，软硬不说，接口起码得有，而且FC的链路层Frame同步，提取功能要有，其他上层逻辑可以用CPU模拟。

Oooops，我的问题描述有误，应该是：**FCoE**中，**FC-2**帧通过以太网承载，**FC**协议处理能否用主机**CPU**实现？按照冬瓜头的解释，我理解此时**FC-2**、**FCoE Mapping**属于“上层逻辑”，可以通过**CPU**实现的。

这样的话是可以的，而且我推测很可能一开始某些FCOE卡大部分逻辑就运行在其驱动程序中。

18. Stripe与Block的关系

在创建**RAID**的时候，会让你选择**Stripe Size** 以及**Block Size**，**Block Size**跟所创建**RAID**容量超过**2TB**后被主机识别有关系。**Stripe**以及**Block**什么关系没有？最好是有个结构图可以说明一下啊。

不知道具体哪个设备，一般来讲**Block Size**是存储内部对磁盘读写的二级单位。**Stripe Size**是**RAID**横跨所有**RAID**组磁盘的逻辑单元。要说有联系的话，**Stripe Depth**的最小单位就是一个**Block**，除了**Block**，还有多种其他被定义的概念比如**Chunk**、**Segment**、**Slice**等，与各个厂家设计和定义有关，需要去看厂家的白皮书。

另外，著名的Windows操作系统**2TB**极限问题，是说**MBR**分区格式下最大分区**2TB**，而不是说系统只能认最大磁盘**2TB**，磁盘可以认到很大，只要注册表中打开**48b**寻址模式即可。解决办法是使用**GPT**分区或者动态磁盘。而且，**Block Size**与这个问题无关。

19. Loop与RAID

如果是阵列里的**Loop**架构，那么**RAID**如何工作呢？如何保证多块硬盘同时读写呢？还有就是这个**LOOP**如何收敛呢？冬瓜兄，我理解**LOOP**的原理是一个节点占用后需要完成操作再释放令牌，磁盘干活不就是磁盘读写和传输么，总得让它一次**I/O**操作结束才释放令牌吧，其他磁盘岂 不是得等着？呵呵，不知道理解是否正确，恭听各位高见。

请求发过去，磁盘总得干活吧，干活的时间，其他盘也给了任务了，就像boss一样，不让人闲着，闲着成本就高，一个道理。呵呵，仲裁权不是这么弄的，仲裁不是一个**Session**。一个写操作底层对应着多个**Frame**，每次尝试发送一个**Frame**，都要仲裁，而不是仲裁一次等这个**IO**

完成的。每个Frame都要仲裁，这样才能保证其他设备不被饿死，如果是Session的话，早就被饿死了，有Session的那是更加上层的协议，比如TCP，底层需要做的是充分保证拓扑上每个点的公平性。另外，以太网也是每Frame都要进行CSMA的，而且估计你也不知道网卡每收到一个Frame，都要中断CPU来执行驱动程序提取这个Frame中的上层Payload，这里顺便告诉你一下。

20. RAID初始化的问题

能详细说明下，做了初始化再使用**RAID**的好处吗？

与实现方法有很大关系。如果所有磁盘在加入**RAID Group**前都预处理过，则不需要初始化。如果没有预处理比如以前用过的磁盘而没有清零直接加入**RAID Group**，则需要初始化。有两种方式：前台独占性初始化，后台初始化。前者初始化结束之前不能接受IO。而后者可以一边接受IO一边初始化，但是后者对IO的性能有很大影响，控制器会在后台连续地对所有条带初始化清零操作，或者读取所有数据盘上的数据进行校验并将校验值写入Parity盘，视不同设计而定。期间一旦遇到上层下发的写IO，则临时跳跃到对应的条带进行操作，完成后再转回来继续。写IO刷盘之前如果对应的目标条带尚未初始化过，那么控制器会直接在内存中就计算好这个条带的内容然后整条写，这个整条中包括此写IO的数据Segment，其他磁盘的全0的Segment以及计算好的Parity Segment。对于RAID初始化的详细描述本附录后文还有。

你的书上说初始化有两种情况：一种是对所有的盘全部写**0**；另一种是保持原来的数据，只是写入校验位。我这碰到的应该是第二种情况，好像没多大必要都写**0**。还有就是我这的**RAID 5**不初始化也是可以用的，是不是对性能上或者安全上有影响啊？还望再解释一下，非

常感谢！

凡是没有写0，拿了盘就加入RAID Group并且立即可以使用的，属于后台初始化，性能会受到影响，因为写入的时候它有必要读出必需的部分来进行校验，几乎都是全条带读出校验，所以这个时候，RAID 5写惩罚将达到最大。

写的时候为什么还要读出以前的旧数据来校验啊？直接写入对应的扇区不就可以吗？还望再解释一下，很想搞懂它的具体过程，谢谢。

请先理解校验型RAID写入时候的具体动作。直接写，其他什么都不干？那还叫RAID 5么，那不RAID 0了么。未初始化的RAID 5，其校验值根本就是不匹配的，因为盘是拿来就用的，上面的数据均未清零，写入某个块，就需要读出整个条带上其他所有块来共同进行校验，并写入正确的校验值。这个条带校验完毕后，如果再有写IO针对这个条带，那么就按照正常RAID 5写入时候的动作。具体请参考《大话存储》，仔细思考。

21. 热点盘是什么东西

存储中他们说的热点盘是啥意思？

热点盘就是几乎每次IO都会读写的那个盘。比如RAID 4中的校验盘，每次写IO无一例外均会读写这个校验盘，使得这个盘的速度成为瓶颈，而且也容易疲劳而损坏。除了RAID机制比如RAID 4导致的不可避免的热点之外，其他因素，比如某个应用程序长时间不停地读写某个盘，而系统中其他的磁盘则相对空闲，则这个频繁读写的盘相比于其他

盘也是热点盘。比如数据库存放日志的盘，相对于存放数据文件的磁盘，就是热点盘。

如何解决热点盘的瓶颈呢？

避免用RAID 4，将需要频繁读写的盘放在RAID 0/RAID 10/RAID 5之上，使这种潜移默化的损害平衡到多个物理磁盘上，降低每盘损坏的几率和提高IO性能。其次，合理设计应用程序。

22. 如何放置数据文件的问题

Oracle的库，其中一个**2TB**，一个**2.6TB**，对应数据和索引文件，从**Oracle**的性能来，最好能放到不同的IO上。**Windows Server 2003**目前使用存储，**RAID 0+1**后总空间是**5TB**多，当前这种方案整个做**RAID 0+1**，然后在**Windows**中分两个区，分别放数据和索引。我的想法，能否在存储上分别做两个**RAID 0+1**，在**Windows**上被认成两个不同的物理盘？IO性能更好一点，这点能实现吗？不清楚存储的原理，这样做性能是否更好一些？

后者更好一些。如果做成一个大的RAID Group，数据和索引的IO会有冲突，只能排队，但好处是空间分配灵活。打成两个RAID Group，各自就有了专用盘，在RAID这一级上就不会冲突，但是在底层比如FCAL上依然是冲突的，但这是无法避免的。一个例外就是两个RAID Group的盘分属不同的Loop，就不会在Loop这层冲突，但是依然会在FC控制器处有冲突的可能。比如同一个FC Chip管理共享的多个AL环，避免方法就是使用每Loop独立FC Chip的卡。然而FC Chip这一层可以解决，但是再下一层也就是IO总线这一层，依然可能冲突，比如多个AL环多个Chip多个FC卡同处一个IO 总线，避免方法是让其分属不同的

总线。再往下就是CPU冲突，避免方法使用多核的、前端口的共享等。顺带说了一大堆，呵呵。

23. RAID 10和RAID 5的详细对比

RAID 10和RAID 5相比，哪个性能比较高呢？

RAID 10比RAID 5在读方面稍微强了一点点，但是强的这一点点并不是因为数据被镜像了所带来的，而是因为RAID 5比RAID 10多了一个Parity盘，无法接受用户数据IO。我们来举个例子如何。8块盘。

RAID 10

11223344

55667788

RAID 5

1234567P

8

有12345678这四个数据放在这个8盘RAID 10上，而8盘的RAID 5，则是1234567P，P代表parity。先来看随机小块IO，比如上层发来这样一组IO，读1，然后3、2、4、8、5、7，此时，抛开FCAL底层不讲，RAID 10之上可以并发这8个IO的，每个盘处理一个IO，此时RAID 5之上只能并发7个IO，也就是1234567，废了一个P盘。所以说读方面，RAID 10的性价比远不如RAID 5。

再来看写方面，同样是上面的RAID组，RAID 10上，抛开镜像过程是异步还是同步，全部按照同步镜像来算，也就是说写入一块盘则必须同步写入另一块盘。假设上层某时刻发来一组写IO，1、6、3、4、5、7、8，RAID 10可以并发的IO只有4个，也就是永远等于磁盘总数除以2。再来看RAID 5，同样这组IO，其可并发的是1~7，构成整条写，写效率大大高于RAID 10了。这个例子貌似对RAID 10不公平，我们先来看单一Segment的IO行为。单个Segment写的情况下，RAID 10之上会产生两个实际Segment的IO，而RAID 5则对应了两个读和两个写的IO，比RAID 10多了两个读。如果是两个Segment的IO，比如1、2，则RAID 10对应了4个写操作，RAID 5此时采用了读改写，即3个读3个写，比RAID 10多了3个读，少了一个写。依此类推，3个IO情况下，RAID 10是6个写，RAID5则是4个读，3个写，比RAID 10少了3个写，多了4个读。至此，RAID 5已经达到读改写和重构写持平态，IO再多可以实现重构写，极端态下可以实现整条写。显然，重构写和整条写就比RAID 10高效多了。

然而，上述结果是基于小范围的随机，也就是一批IO很大几率都分布到同一个条带中。如果上层的随机IO离散度过大以至于分布到同一个条带的几率大大降低，此时，RAID 10的优势凸显，因为不管IO离散度如何，RAID 10始终可以并发半数磁盘数的IO。而RAID 5此时就要碰运气了，可并发几率大大降低，此时RAID 5性能与RAID 10相比就差别比较大了。

综上所述，相同情况下，RAID 5的写操作总是小于等于RAID 10写操作，而读操作总是大于RAID 10，在高度随机写IO的情况下，RAID 5读改写的几率最大，所以相比RAID 10多出了很多读操作；而在随机性不太高并且并发几率比较高的情况下，RAID5重构写甚至整条写的几率

大大增加，此时RAID 5效率与RAID 10接近。加上读操作比写操作成本和效率更低，而且如果考虑入Cache的预读效果加成，总体来讲RAID 5与RAID 10实际使用中，在小范围随机IO的情况下效果差别不是很大，如果考虑性价比，首选RAID 5。如果写IO离散度过大，此时RAID 5性能下降就比较厉害了。优化的控制器会平衡IO Queue的大小，Queue越大，就有更大的几率优化成并发IO，但是延迟和开销却越大；Queue越小，优化成并发IO的几率越小，然而开销和延迟就越小。

24. 磁盘数量太多的RAID组，有什么优缺点

磁盘数量太多的RAID组，有什么优缺点？

一个RAID组中的磁盘数量越多，IO性能就越高。当然控制器起到至关重要的作用，有些控制器的硬件规格不能承受太多磁盘同时IO，总有限。通常情况下，一个RAID组推荐磁盘数量为8~14块。磁盘太多也有缺点，比如扩展性差，因为RAID组一旦创建完成就不可以剔除其中磁盘，拔掉任何一块磁盘，便处于降级状态，危险增加，必须插入新磁盘来顶替。所以创建RAID组是不可逆过程，如果一次就用了太多磁盘，今后余地就很少了，尤其是拥有磁盘数量不多的情况下。还有一个缺点，就是Rebuild过程会很长，Rebuild需要读出整个条带的数据，如果磁盘数量太多，则Rebuild时间会严重增加，系统不堪重负。

25. 多主机共享LUN的问题

我们公司用HP-MSA2000磁盘柜存储8TB的数据，有三台Dell的1950服务器（操作系统为Centos）访问这个存储柜，存取数据。连接方式是服务器通过光纤交换机直连磁盘柜，然后挂载到本地访问。问题：从其中一台服务器往磁盘柜上传文件，这个时候其他服务器从本

地挂载的硬盘上看不到刚才上传的文件，除非是先卸载掉，然后再挂载才能看到。这是为什么呢？另外这种连接方式是不是**SAN**呢？

多主机共享同一个LUN，除非所有主机上都安装集群文件系统或者**SAN**共享仲裁软件比如**SANergy** **ImageSAN**之类，否则会出问题。问题的根源就在于三台主机上的文件系统写入LUN的信息不是同步的，而是各有各的缓存，写入的数据先进入缓存。而且也没有锁机制，致使数据不同步不一致。要达到你的目的，多主机共享数据，要么用**NAS**要么用特殊的文件系统，但是推荐用**NAS**，成熟，方便，只不过带宽可能满足不了要求。**SAN**共享软件当然所有节点都得装。不知道**Reiserfs**本身是否支持共享，不支持的话，上面已经回答了。至于**DAS**还是**SAN**的，不重要。

而这方面我自己的定义是这样的：广义来讲，**SAN**指的是存储网络和其中的所有元素，也就是说，只要通过网络来传输存储数据的，都属于**SAN**范畴。**NAS**也属于**SAN**。**FC**直连也应当属于**SAN**，因为**FC**是一个包交换的现代网络传输协议，不管物理拓扑上是直连还是通过交换设备，其协议本质已经是**Network**。而目前的存储架构中，唯一排除在**SAN**范畴之外的应该是传统的**SCSI**线缆、**SATA**线缆、**IDE**线缆、**USB**、**1394**等访问方式。因为它们不具备包交换的特性，如果硬说它们也是网络的话（其实就是网络），那么也是一个共享总线的网络，不具有次世代网络特征。然而，如果从最大的角度来看，目前所有存储架构都属于**SAN**，从一开始就属于**SAN**。

26. 如何利用多以太网链路达到最大带宽的问题

环境描述：服务器端安装了多路软件，**iSCSI**磁盘阵列只有一个**LUN**分区，控制器上面有**4**个**iSCSI**主机接口。假设此款磁盘阵列的每

个主机接口只有**100MB**的带宽，连接到千兆交换机，此款交换机支持链路聚合功能，即**Port Trunking**。即把四条百兆链路聚合成一条总共**400MB**带宽的链路。磁盘阵列的这个分区根据四个端口都映射给了服务器，即服务器在没有配置多路径软件时可以看到四个完全一样的分区。使用了多路径软件之后，系统只有一个分区，即任何一条链路出现故障，不影响对磁盘阵列的正常读写。问题：

（1）服务器向磁盘阵列读写数据时，是否能够通过链路聚合而达到**400MB**的带宽？

（2）多路径软件只认可一条链路，主要功能是容错，然而，是否在读写时只使用一条链路把数据写入磁盘阵列呢？假设此磁盘阵列本身不支持链路聚合的功能。如果支持链路聚合功能的话，直接在磁盘阵列把四个主机接口绑定成一条链路，那什么问题都解决了。

第一个问题：这种情况，只有在同时满足下列几个条件的情况下，才能达到**400MB/s**。

- 多路径软件支持多路径并发。
- 盘阵端的每个以太口必须都设计为独享带宽，不允许有任何一个与其他端口共享带宽，也就是不允许两个或者几个端口在一个HUB上。我说的带宽是说盘阵内部背板上的带宽，而不是说外部的交换机或者HUB。
- 客户端的IO类型，随机小Size的IO，达到**400MB**不可能。需要连续大块的并发IO。
- 后端前端中部总线以及处理能力均满足**400MB/s**的需求。

值得一提的是，利用多路径达到的聚合，是原生的，不需要盘

阵“支持”，但是盘阵一定要支持同时向多个前端口映射这个LUN。

第二个问题：多路径软件一般是支持多路径并发的，不过效率很少有满载的。可以单Active也可以多Active。单Active，最大带宽只能是100m。多Active，带宽恐怕也达不到400m，理论值而已，实际上很难达到。

多路均衡在OSI的多个层次上均可实现。

链路层：Nic Teaming，Trunking/Channelling。

网络层：根据IP地址和MAC地址的均衡。

传输层：用Multi TCP Connection进行均衡，每个TCP连接使用单独的IP和网卡出去，达到对端虽然是不同的IP和网卡，但是上层给屏蔽掉了。一个例子就是iSCSI中的Multi Connection per Session。

应用层：利用多路径软件进行均衡。

所以说，达到多少带宽，使用什么方法，在哪一层实现，都会对结果产生影响，而所有这些方法的本质皆相同。

27. 关于LUN切换的问题

大家好，我有一个问题向大家请教。环境是：一台双控磁盘阵列，分别直连两台服务器，有两个疑问是：

(1) 正常是应该用A控管理LUN1，B控管理LUN2，但我划分LUN的时候把LUN1分配给了B控，结果也能在服务器上看见LUN1，请问这种情况正常么？

(2) 我按正常分配**A**控管理**LUN1**，**B**控管理**LUN2**，如果**A**控制器坏了，是不是**B**控能自动接管**LUN1**？这种情况还用在服务器上装多路径软件么？

脱离了实际设备，这个问题就没有确切答案。有的在盘阵端支持**LUN**切换到活着的控制器，而主机端不需要切换，当然主机端需要有其他程序来控制这种行为。有的则全靠主机端多路径软件，所有控制器都可以映射所有**LUN**，具体问题具体分析。

28. 关于星形拓扑、FC拓扑的问题

我记得你的书里面有提到光纤的星状拓扑，不知道和**HDS**高端产品的**Hi-Star**光纤交换，还有**EMC**高端的光纤矩阵是什么关系。搞混了拎不清楚，麻烦请教下。

FC拓扑中的星型拓扑，和盘阵控制器内部CPU内存外设之间的拓扑，好像是两码事吧.....我一再强调不要再用什么“光纤硬盘”什么“光纤交换机”了，这里又来了个“光纤矩阵”，本来已经很不容易入门，如果再搅浑水，会误导很多人，甚至自己都糊涂了。**HDS**高端存储内部控制器总线类型多少年未变，全部使用Crossbar交换矩阵芯片，高成本高性能。**EMC**的高端存储内部总线架构则使用点对点直连结构，非Crossbar交换矩阵。

29. 一些专业术语的理解

小弟刚接触存储设备，一次面试过程中不知道以下这些专业术语是什么意思，请知道的兄弟们解释下，谢谢！快照代理，虚拟快照，分裂镜像。

快照代理：泛指安装于OS中的一个程序，可以与FS或者应用发生交互作用，使得Snapshot开始之前FS缓存中的内容被写入硬盘或者应用层触发一系列动作，让FS和应用程序产生一致点，避免恢复Snapshot后不必要的fsck过程和应用级恢复过程。

虚拟快照，不懂。你可以反问他这是谁家的定义。

分裂镜像：对某份数据做同步镜像，空间双倍占用，当需要这份数据某时刻的Snapshot的时候，Freez IO，然后解除镜像关系，脱离开的镜像就是这份数据此刻的一份Snapshot。这时你可以回答，嗯，这么看来，分裂镜像产生的Snapshot是实Snapshot，而通过复制MetaData Tree产生的Snapshot就是虚Snapshot，或者就是上文中的所谓“虚拟快照”吧。

30. RAID 5的读写步骤

RAID 5的读写步骤是怎样的？

这个Internet上都可以搜索到的。分4个步骤的是写，而且写也是分情况的，不一定是4步。随机写的情况下，写惩罚严重，先读出被覆盖的数据和Parity数据，然后用待写的数据和被覆盖的数据计算新Parity，然后写入新数据和Parity。就是这样。

31. SAN共享问题

有个问题，书中说到**NAS**可以提供同时对同一个目录或文件夹的访问，而如果是**SAN**网络，如果共享同一存储空间，需要借助软件。如果把**SAN**后端存储中设置为只一个人可写，其他只读，不是也称之

为多台客户端在一台存储中共享访问么？（当然了，虽然具体的访问模式不一样。）

一写多读理论上是可以的，不过需要在特定情况下。比如所有客户端统一使用Direct IO+Write Through模式对目标进行IO，但是DIO+WT模式并不能保证客户端缓存的文件Metadata也同步被写入磁盘，如果目标文件的Metadata对程序没有什么影响的话，那么这种方式是可以满足要求的。但是现实中很少这么实现的，因为大多数情况都是多写多读的，不只读是应用的事情了。如果不使用DIO和WT模式，就算用只读mount，对方写入之后，我方也不能得到最新的数据，除非等下一次FS从磁盘提取最新的inode tree到buffer，这样会造成不一致的。如果写方和读方之间有其他上层关键业务逻辑，数据不同步造成的后果可能是极其严重的。

32. 如何判断造成邮件队列堵塞的原因？

这是我上网看到的一个方案 根据作者列出的4点可能造成邮件队列堵塞的现象，感觉不是一个NAS就可以解决这四个问题，方案如下。

1) 用户现状

某ISP信息网原邮件服务器采用某公司小型机及其磁盘阵列组，邮件服务器每日处理邮件能力不足3000封，每日早晨上班和下午下班使用邮件高峰都会出现发送邮件队列堵塞。

2) 需求分析

造成邮件队列堵塞的现象可能因为：

(1) 小型机无法同时响应来自磁盘阵列盘上的邮件分发工作。

(2) **UNIX**系统的文件管理服务在文件读写操作过程中，占用**I/O**资源无法释放，小型机处理并发**I/O**进程的能力有限。

(3) 即使网络带宽充足，几千封邮件排队等待处理，也会造成堵塞现象。

(4) 另外，网络带宽不够，还可能造成数据在网络传输中的读写瓶颈。

3) 系统设计

购买一台**NAS**存储设备作为原服务器的数据存储设备，将原服务器中的电子邮件数据全部转移到该设备上，保持原服务器继续作为电子邮件服务的应用服务器。此时，一切问题迎刃而解。

原作者的方案点评：

NAS产品采用了专用技术优化了文件访问的**I/O**效率问题，提供了比较理想的文件访问能力。尤其是**Veritas NAS**解决方案还提供了容灾和故障管理能力，使得**NAS**技术也可以成为容灾系统的组成部分。

个人理解：

(1) 如果原来的磁盘阵列只用于存放电子邮件数据，将原服务器中的电子邮件数据全部转移到**NAS**，这样做，原来的磁盘阵列是不是只能用于其他需求了？

(2) 保持原服务器继续作为电子邮件服务的应用服务器。（这里

的服务器应该不是前面说的小型机吧（会拿小型机做服务器吗）？如果不是，购买**NAS**后，**NAS**是不是取代的是原有的磁盘阵列呢？）

（3）优化了文件访问的**I/O**效率问题，提供了比较理想的文件访问能力。（小型机配置没有提升，只是买了台**NAS**，这样小型机就可以同时响应来自**NAS**上的邮件分发工作了吗？这么说原来的颈瓶不就是出于原来的磁盘阵列不能提供一个好的文件访问能力了吗？可作者说的是小型机无法同时响应来自磁盘阵列盘上的邮件分发工作。怎么感觉是小型机性能跟不上呢？）

（4）网络带宽不够（增加一台**NAS**带宽就够用了吗？）

太多不理解了，希望各位不要见笑，新人阶段，各位能解析下吗？

（1）如果原来的磁盘阵列只用于存放电子邮件数据，将原服务器中的电子邮件数据全部转移到**NAS**，这样做，原来的磁盘阵列是不是只能用于其他需求了？

可以将数据文件分别放在原来的阵列和**NAS**中，平衡负载。

（2）当然还是原来的机器。不拿小型机做服务器难道用笔记本做？

（3）这个方案，一点根本的调查分析都没做，只是给出了几个猜测，也都在理，其实只要在小型机上用各种命令查看输出，就能初步推断到底瓶颈出在哪里。一点确切结论都没有，把所有可能摆上了，然后就说换**NAS**。他也没提到以前的到底还用不用，新**NAS**是分担负载还是全盘替换过来。而且，就算换了**NAS**，也不一定会有改善，也得看**NAS**

的规格、后端带宽和IO需求。

(4) 他说的是前端带宽也许不够，都是推测。后端存储带宽他没有调查，不明确。

33. 随机和连续IO的问题

随机（**random**）和连续（**sequential**）读写的区别，是对**Volume**来说连续的读写还是对物理硬盘来说连续的读写？

是哪个层面的“磁盘”，物理磁盘么？一般来讲就是根据一定算法，比如前几个是连续地址，后来这个差太大了，就算**random**了。至于哪个层次上的，哪个层次都有。但是最终影响性能的还是要看最底层也就是物理磁盘，因为最终是磁头臂的寻道影响了性能。有意义的就是最终物理上的地址。关于各种IO属性的总结和联系，请阅读《大话存储》相关章节。

34. 负载均衡的方法

有一个**SAN**环境，有几台存储管理服务器，现在想实现**IO**负载均衡，就是几台服务器用一个虚拟**IP**给用户，用户有数据**IO**时，自动找到**IO**负载较小的服务器，有些类似于**LVS**的思想。不知道有没有实现这个想法的软件或方法。

LVS不就是么？硬件比如**F5**、**Packteer**之类的。都是基于**TCP**及以上层次的，高级点的可能还会识别应用层内容来均衡。如果你的意思是说仅仅均衡存储的**Block IO**，比如用一个前端设备接收主机**IO**，然后后端向多个存储设备分发负载，这种思想是存在的。但是这种物理拓扑目

前是不存在的，真正体现这种思想的拓扑是存储集群，比如NAS集群，集群中并没有一个前端Gateway设备，而是多个节点共同提供访问。它们有一个虚拟IP地址用于接受前端IO，而且负载均衡并不是随机分发，而是按照预先设定好的目录来判断对应数据IO应分发给哪个存储设备。数据一旦存储之后，是一直都在固定的地方的，当然一些带有Tier数据迁移功能的设备除外，不管是否带有迁移功能，数据被写入后，绝不是可以随机乱跑的，所以要做到IO随机分发是不可能的。随机分发方式的负载均衡只能在相对上层来做，将所有数据人工分目录分类按照访问频率和要求存放在多个后端存储设备上来达到人工的均衡。

35. 单个目录下的文件数量和性能问题

为什么单目录下子文件过多会影响性能？如1个目录下有**10000**个子文件，那么读取某个文件的速度将会明显慢下来，这和文件索引有关吗？索引中如何组织这些节点？谢谢大家帮忙。

是的，与索引有关。1万个不算多。上百万个就看出来了。

上百万个慢是文件系统整体吧，那么和当前目录怎么关系上呢？

一个差不多的文件系统支持几百万个文件不算什么。我是说一个目录下，不分子目录，直接放它几十万上百万的文件，此时检索这个目录索引很费资源。支持数量有限是因为目录这个对象本身容纳的大小有限制，目录就是一个容纳文件名和文件对应inode号的容器，被限制了，那么容纳的条目也就被限制了。读取某个文件速度没影响。但是查找就费劲了。某些文件系统的索引机制不完善，甚至没有什么优化算法，致使每次查找耗费更多的时间。

36. Spare的数量与风险的关系

一个DS 3200，打算建一个RAID 5，本来设2块hot spare的。后来由于容量限制，想改成一块，这样会增加风险吗？

2块是为了防止Rebuild结束之后过一段时间又坏了一块，能坏第一块就证明其他的也离坏不远了。2块同时坏的几率是比一块小。一旦某条件达成，坏的不只是2块了，说不定坏一批或者HBA问题直接Loop上的全不认，真这样的话，多少块spare也没用，一块就够了。所以需要综合考虑。

37. 接口带宽与磁盘传输率的关系

1000M带宽满跑流量是多少？SCSI硬盘的最大传流量是多少？在不考虑其他因素的情况下.当两者满跑的时候，谁先成为瓶颈？

1Gb的跑满是120MB（包含协议开销）。iSCSI硬盘，没有这种硬盘。有iSCSI盘阵，多块盘一起做成LUN，这样的话，大块连续IO瓶颈在网络，小块随机IO瓶颈在磁盘，不过也得具体分析，与磁盘数量，控制器等都有关系。理想状况下100~110MB/s是没问题的。

38. Page Cache作祟

有一个卷lv1，块大小为4KB。此时我用dd做写测试，`dd if=/dev/zero of=lv1 bs=4k`，当测试的bs为4KB时，数据正常。当把bs的值减小到4KB以下，发现读和写都有数据，而且流量相当，不知道是什么原因。也就是，不理解当测试的块大小“小于”卷的块大小时，就会产生读的数据。

呼呼，正常现象了。lv1是块设备，你用的什么os？块设备你对它写小于4KB的数据，它要先读出来，然后写进去。用裸设备就不会出现这种情况，1KB、512B都可以。Page Cache在作祟。

39. 关于标称带宽与实际带宽的问题

简单的说，**DS 4700**的机器用两个**FC 4Gbps**的接口，怎么样才能让它可以提供**8Gbps**的主机连接带宽？

估计很难达到。4700标称后端带宽没记错的话是1.6GB/s，而且实际中一半估计都达不到，0.83GB/s不可能达到的。

40. 卷共享的问题

A机和**B**机都接在**SAN**上面，操作系统是**Redhat Linux AS 4.5**；存储是**HP**的设备。在**A**机上写的文件，在**B**机上不能访问，请问有无解决办法？需要在磁盘阵列上做什么设置么？谢谢。

共享同一个卷，请小心，弄不好就数据损毁，得用集群文件系统，例如Stornext，或者集群卷软件Sanergy之类。还有一个野蛮办法，使用DIO+WT模式，不过只能保证弱一致性。

41. Proactive Copy与Rebuild的关系

热备盘的前摄性模式和重建模式什么区别？

可能这就是英文Proactive吧。一旦发现哪块盘不对劲，Prefail，复制数据，盘对盘，而不是Rebuild，这样速度快很多，性能受影响也很小。

42. Loss of Sync的含义

Loss of sync过高意味着什么？

意味着物理层两端时钟或者接口接触等方面有问题。这种是可以自行恢复的，恢复不了就是link fail了。

43. 数据删除之后如何恢复？

一块**SCSI 36GB**硬盘，作为**SYBASE 12.5**数据库的数据文件设备，最近出现读写错误，估计是硬盘有坏块，做了**FSCK**后问题依旧，数据文件无法备份也无法复制，昨天一哥们来通过数据库倒数据，结果误操作，把原来数据的表全部删除后，又重新创建，我现在都哭了，大侠们看看能恢复到误操作前的数据状态吗？

得找数据恢复公司了。重新创建的，往里填数据了么？没填的话可能只是hole，还有希望恢复的。

44. FC阵列是否可以直连的问题

FC磁盘阵列是否可以直连？

可以直连，只能以FCAL模式直连了。因为用交换机的情况下，各个节点都是N端口，都需要进行Flogin等一系列FC逻辑，直连情况下，如果两端都还是N端口，则无人会响应Flogin。FCAL则是各个节点自初始化自选举自仲裁的，所以可以直连。

45. RAID 3的IO问题

RAID 3 每进行一笔数据传输，都要更新整个**Stripe**，即每一个成员磁盘驱动器相对位置的数据都一起更新，因此不会发生需要把部分磁盘驱动器现有的数据读出来，与新数据做**XOR**运算，再写入的情况。这种情况在**RAID 4** 和**RAID 6**中会发生，称之为**Read、Modify、Write Process**，即读、改、写过程。因此，在所有的**RAID**级别中，**RAID 3**的顺序写入性能是最好的。怎么理解？

RAID 3任何情况下都保持整条写，解释完了。其实上面已经解释的很清楚了，不理解的话表明对更加基本的一些底层概念还没有理解，所以请再阅读《大话存储》。

46. 脑分裂是怎么回事？

谁能帮我解释下到底**Split-brain**即脑裂是怎么一回事？

就是说，本来一个大脑的两半球互相配合，变成了分裂成两个独立的大脑，都认为对方已死。此时，双方都尝试接管集群资源，造成冲突，后果严重。解决办法：使用**Vote**磁盘仲裁，利用对特定数量的磁盘放置**SCSI reservation**的方式来决定谁最终掌管集群资源，或者，最极端的**Powerfence**，谁抢先关掉对方的电源谁就获胜。

47. 硬盘的**Block**与**Extent**的区别

硬盘中**block**和**extent**有何区别？

这些名词都请参阅具体厂家的定义。硬盘上只有**sector**，还有**track**。**Cylinder**也是由**track**组成的一个逻辑定义。其他什么都没有。再往上的都是逻辑结构，并非物理结构，牵扯到逻辑结构，那就和定义者

有关了。

48. 一些IO测试结果的分析

先介绍一下大概情况：阵列是**EMC的CX-310**，全光纤盘，与主机连接用的是**4GB**的光纤通道卡，做的**RAID 5**。机器是**IBM System x3650 (7979B9C)**。服务器的硬盘是**SAS**盘。操作系统是**Centos5.2**，文件系统是**ext3**。下边是用**dd**测出来的速度（偶尔会更低）：**sda**是本地**sas**盘。

sda

写

```
dd if=/dev/zero of=/app1/test/dd.log
# dd if=/dev/zero of=/app1/test/dd.log
1975913+0 records in
1975912+0 records out
1011666944 bytes (1.0 GB) copied, 15.0658 seconds, 67.1 MB/s
```

读

```
dd if=/app1/test/dd.log of=/dev/null
# dd if=/app1/test/dd.log of=/dev/null
1975912+0 records in
1975912+0 records out
1011666944 bytes (1.0 GB) copied, 6.53719 seconds, 155 MB/s
```

读写

```
dd if=/app1/test/dd.log of=/app1/test/dd2.log
# dd if=/app1/test/dd.log of=/app1/test/dd2.log
1975912+0 records in
1975912+0 records out
1011666944 bytes (1.0 GB) copied, 17.1978 seconds, 58.8 MB/s
+++++
```

阵列

```
/home/sakai/mount/bak
```

写

```
dd if=/dev/zero of=/home/sakai/mount/bak/dd.log
# dd if=/dev/zero of=/home/sakai/mount/bak/dd.log
3544417+0 records in
3544417+0 records out
1814741504 bytes (1.8 GB) copied, 24.3567 seconds, 74.5 MB/s
```

读

```
dd if=/home/sakai/mount/bak/dd.log of=/dev/null
# dd if=/home/sakai/mount/bak/dd.log of=/dev/null
3544417+0 records in
3544417+0 records out
1814741504 bytes (1.8 GB) copied, 11.9376 seconds, 152 MB/s
```

读写

```
dd if=/home/sakai/mount/bak/dd.log of=/home/sakai/mount/bak/d
```

```
# dd if=/home/sakai/mount/bak/dd.log of=/home/sakai/mount/bak
3544417+0 records in
3544417+0 records out
1814741504 bytes (1.8 GB) copied, 26.4129 seconds, 68.7 MB/s
```

569MB速度显然是Cache速度，既然读入的是FS文件，那么显然FS缓存在作祟。umount然后mount（或者# echo 3 > /proc/sys/vm/drop_Caches），然后再测试你就会发现根本没有500多MB。至于155MB/s的速度，与很多因素有关，如IO调用方式、IO Size等。centos不熟悉。dd在Linux下一一般都是sync同步IO调用，每次IO Size=4KB，此时除非同时运行多个dd，否则只一个dd，155MB速度也算是不错了！

df 查看的是文件系统使用率，磁盘使用率如何获取呢？大家谈谈思路也行，谢谢。

这是个匪夷所思的问题。如果拿一块崭新的磁盘，你可以说它使用率是0，一旦使用了一段时间，就不好说了。只有其上“有用”的区域，才能算是使用率，如何计算“有用”的区域呢？就是看文件系统。

文件系统之外的有两部分：操作系统其他逻辑层比如VM，在disk上写的数据；磁盘控制器比如RAID控制器在磁盘某些区域写入的数据，这些也算“有用”的，但是不能通过FS来查看了。

这个问题脱离了上层的卷管理软件或者文件系统，是没有意义的。磁盘本身并不知道自己身上哪些是正在被使用的，哪些是没用的。“有用”和“没用”是FS或者VM管理的，VM用磁盘，FS用VM，应用程序用FS，人用应用程序，如果人说一句，这些数据都没用了，扔了吧，那么

不就是都没用了么？顺便说一句，FS里面都有bitmap来记录对应存储空间上的每个扇区或者簇区是否被某文件占用，占用则有用，未占用则没用，也就是空闲。空闲的扇区不一定里面都是0，可能是原来数据的Zombie，这个请阅读《大话存储》相关章节。其他的OS不谈，在AIX里面应用能使用多少硬盘空间在lspv里面可以看到（Free PPs+Used PPs=Total PPs）。Free PPs是还可以使用的空间，而Total PPs就是总空间。这个总空间肯定比磁盘的裸空间要小，但通常相差不是很大，通常就是相差一个PP的大小。所以我说如果不严格要求的话可以近似用lspv来看。不过如果你说的是磁盘阵列上物理磁盘的利用率，那我们说的就不是一回事了。说白了，VM就是个粗线条的FS。最终知道使用率的，除了FS就是VM，最终物理Disk，没法知道。

49. NAS控制器和服务器的区别

存储设备是通过NAS控制器挂到网络上的，NAS控制器也被称为瘦服务器，实践方面接触的比较少，谁知道这个NAS控制器的具体结构请介绍一下，主要是和一般的服务器的区别（从结构上谈，功能上网上搜的到）。如果可以，其他方面的也谈一下，比如具体的工作流程。

NAS就是一个提供文件服务的设备，后端既可以使用自己本地硬盘，也可以连接到SAN设备上拿硬盘。之后上面加一些虚拟化的层，比如Volume、LUN，然后必须再添加一层本地文件系统层，最终一层必须是网络文件系统，比如CIFS、NFS等。就是这么个架构。

50. 关于RAID信息是否写盘的问题

我知道很多RAID控制器为了好管理都会写类似超级块的信息到组

成**RAID**的硬盘上。我想知道有没有一种控制器除了上层操作系统的写数据外是不会写任何信息到硬盘上的。

稍微高端一点的**RAID**控制器好像没有哪家这么做，原因很简单，一是卡上没有足够的地方存放，而是写到硬盘上，卡坏了，换个卡，载入，照样用。后者是根本原因。早期的**RAID**卡甚至不允许磁盘槽位错乱，很有可能就是因为它没向磁盘上写入对应的信息，而只是在卡上自己保存了一份，磁盘一旦错乱，**RAID**就不认了。写到磁盘上之后，磁盘不管放到哪个槽位，逻辑上的结构依然没有变，依然可以组成**RAID Group**。

51. 测试速度达不到要求

曙光磁盘阵列柜，挂到服务器上用**HD Tune**测试只有**5MB**。

提供可能的原因：

- (1) IO不对齐或者其他原因导致的内部惩罚。
- (2) 用**IOmeter**试试。
- (3) **Queue Depth**太低。
- (4) 用多线程并发访问，不会只用了一个线程吧？
- (5) 也是最有决定性的原因，IO属性是什么。

52. **FCOE**和**iSCSI**的区别

请问下**fcoe**和**iSCSI**的区别是什么？

用于存储网络时：

FCOE: SCSI over FC over Ethernet

iSCSI: SCSI over IP over (any kind of link that support IP)

其实，底层链路不管是什么，都可以支持IP。支持这个词似乎用得不好，但是链路层帧中应该有个字段来描述其承载的上层协议是什么，这就是所谓的支持。IP和链路层位于OSI不同层次，之间可以适配的，所以，不管怎么连通，用什么方式连通，其上都可以盖一层IP来进入TCPIP，这就是OSI的精髓。另外，FCOE现在已经远非这么简单了，FCOE已经逐渐脱离传统以太网基础架构，适配器，甚至交换机都需要专用的，这还谈什么OE呢？本来就是为了和以太网基础架构结合，降低成本，现在却越发走向不开放的道路，个人不看好。

53. 判断IO瓶颈

现有一台**LSI 3994**磁盘阵列（**IBM**的**4000**系列即是**OEM**它的），**2**台**HPDL380G5**与这台磁阵相连，这两台服务器各自有自己的**LUN**，不做双机，只是**2**台服务器一起给磁阵写数据，写到各自的**LUN**上。但是现在只要使用业务，**2**台服务器开始给磁阵写数据，**2**台一共大概有**180MB/s**的写入量。这时读磁阵上保存的数据时，速度很慢，大概**1MB/s**多的速度吧。这台服务器的主机通道是**4Gb/s**的，应该有**400MB**的读写能力，怎么会这么慢呢。服务器使用的是**Windows 2003**操作系统。服务器分别直接连接两个控制器，没有使用光纤交换机。磁阵已经设置为**4G**的模式，磁盘条带化已经为了适应视频而调节到最大值了。

呵呵。兄弟，看来你这瓶颈没出在网络上。虽然用了80个500G 的SATA，但是控制器此时的行为不好说是否都用在了写数据上。180M的写IO，请问这些IO的属性估计一下是IO密集的还是IO Size很大的？如果是前者，具体多少？有没有达到IOPS瓶颈？其次，IO的时候，不一定就真正分摊在了这80块盘上，而很有可能只用到了其中几块盘。最好能用盘阵监视工具看一下当前的磁盘IO情况，CPU和RAM情况，综合分析。读的时候用什么读的？读的多大的文件？RAID类型是什么？这些都要心有一幅图，不然没法做下去的。

54. 网卡瓶颈

在由服务器网卡到交换机再到磁盘阵列的传输链路中，瓶颈在哪里，值是多少？

假设：

（1）网卡为普通千兆网卡，且服务器只有这一块网卡用于连接**IP SAN**中的交换机；

（2）交换机为全千兆全线速转发交换机；

（3）磁盘阵列通过对硬盘做**RAID**能够从出口对交换机提供足够的传输带宽。

那么瓶颈应该可能在两个位置：

（1）服务器网卡，千兆网卡理论带宽**128MB/s**，然而它的实际工作带宽能达到多少？有**60%**吗（**77MB/s**）？如果除去**IP**协议的开销，还剩多少？

(2) 交换机端口传输带宽，千兆的端口，最大实际工作带宽能达到**128MB/s**的**60%**吗？

我想准确地知道这个值，因为我想知道，在这种情况下的**IP SAN**与服务器本地硬盘相比，在传输性能上有没有优势。本地硬盘按**SATA**平均内部传输率**60MB/s**来算。

我觉得为何这帖子至今无人回答，是因为没法回答。你也说了，瓶颈点一在网络，二在磁盘，就这两个。磁盘外面是清一路的千兆通路，所以算一个瓶颈点。盘阵网口后面是控制器OS以及其他硬件，这里算是个潜在瓶颈点，但是再烂，125MB/s的速度也至少能提供，所以本案，此处没有瓶颈。再就是后端磁盘了。还是那句话，再烂，一百来兆的速度起码也能给出来吧。综合分析，您这系统瓶颈将会在网络。

而实际中只要是大带宽需求的环境，确实很大比例都出现在网络瓶颈上，1Gb/s的速率实在是很低了。如果IO Size很大的话，106MB左右是正常的，这也是我平时所见的。IO Size很小的话，五六十兆甚至十几兆都有可能，得根据IO属性综合判断。实际中可能发生各种问题。有客户甚至测试过，用盘阵不如用USB硬盘盒。性能分析一定要有如下条件：①什么应用；②什么盘阵，内部行为如何设计的，CPU和RAM如何；③什么IO，读写，大小，随机程度，QueueDepth并发程度；④什么RAID类型，具体参数，Strip、Block之类，并发度怎样；⑤各种适配卡线等是否底层缺乏兼容性或者参数不匹配。

55. IO是怎么定义的？

(1) 怎样的操作才算是一个**IO**操作？什么操作才算一个**transaction**？

（2）Transaction和IO有什么关系？

你好，关于如何才算一个IO的问题，我可以在这里简要讲解，不过更详细的内容请参考本书前几章。IO在不同层次有不同的概念和单位。一次IO就是一次请求，对于磁盘来说，一个IO就是读或者写磁盘的某个或者某段扇区，读写完了，这个IO也就结束了。

至于transaction，就是更高层的内容了，transaction往往与业务逻辑有关系。比如你去银行存一笔钱，你存这笔钱的过程中，服务器向数据库中写入的所有关联的操作就算是一个transaction，而完成这一个transaction，往往对应了底层对磁盘的多次IO。比如，读出数据库中原来的数据，比如你原来存款是10000元，他读出来了，然后显示在柜台终端上，然后操作员存入5000元，数据传输到数据库服务器，数据库服务器在内存中更改这个数值，从10000更改为1.5000，然后数据库Flush的时候，将对应的数据库写入磁盘，完成后，柜台终端显示成功。这个过程中对应了多次磁盘IO。

补充几点：IO类型有多种，数据型IO和非数据型IO。前者是指IO请求中包含读写扇区的数据的，后者是指IO中不包含扇区数据，而是承载其他信息的，如SCSI协议中的很多操作码，比如0x01就是zero指令，命令磁盘自行向所有扇区中写0。或者诸如report LUN这种常见的指令，它们是命令磁盘做一些其他的动作，而不是真正的读写扇区中的数据。对于网络文件系统来说，也有数据型IO和非数据型IO，前者比如读写操作，后者比如NFS中的mount、fsinfo、fsstat、getattr等。这就是transaction与IO的关系。

56. Cache与IOPS的关系

增大阵列的**Cache**能对**IOPS**有多大的影响？对于顺序读\写、随机读\写影响会有多大？

Cache对于连续IO具有较好的效果，尤其是连续小块IO。针对**Cache**，不同的产品有不同的算法，但是大部分是众人皆知的算法。连续小块IO效果较好的原因是因为预读程序会预先读入与本次IO相连的周围地址的数据，而IO的Size越小，并且IO越连续，性能提升的效果就越明显了。

连续大块IO为何效果不明显？其实这里所说的明显与否是相对的。连续大块IO的时候，更多的压力在于磁盘和通道带宽，**Cache**只起到一个缓冲作用，并没有体现出任何优化的效果。大块连续IO情况下，**Cache Eject**率非常高，系统没有必要也没有这个精力去再做什么优化，也没有可优化的地方。**Cache**在重复读的情况下效果最佳。所谓重复读，也就是外部频繁地读取某些地址的数据，而从来不更改它们。这样，这些数据就会长时间地停留在**Cache**中而无须访问硬盘。

随机IO的情况下，**Cache**效果最小。随机IO的瓶颈在于磁盘寻道，并且更加考验一个产品针对IO的优化程度。但是广泛来讲，随机IO性能低下的问题在机械硬盘的世界里，永远不可能有本质上的解决办法。只有SSD或者其他概念的存储方式才可以从本质上解决随机IO的性能问题。但是面对随机IO，**Cache**也不是一点效果也起不到的，比如某种算法，类似数据库的优化方法，即让数据在磁盘上的分布单位加大，每次读都读出整个extent来碰运气。也就是类似这种思想，比如有一群蚂蚁，你要把它们全收集起来，而你的筷子每次夹的太少，麻烦费力低效，而如果你换成勺子，直接将蚂蚁和泥土一起挖起来，同样可以达到收集蚂蚁的效果。也就是用大炮打蚊子，炮弹爆炸了，那肯定周围的蚊子也被烧死了。

连续读：对于小块连续读IO，Cache优化效果最好；对于大块连续读IO，优化效果不明显。

连续写：对于连续小块写IO，如果Cache是write back模式（通常都应当是write back），由于IO Size比较小，Cache剩余空间充裕，不至于频繁引起Cache flush操作，加之系统不繁忙，有更多时间去进入算法流程对这些IO进行优化调度重排等，所以此时Cache效果良好。对于连续大块写IO，Cache flush频繁，整体瓶颈归于磁盘，Cache优化不明显，仅作缓冲之用。

随机读：瓶颈归于磁盘，Cache优化不佳，只作为缓冲之用。

随机写：对于随机小块写IO，如果Cache是write back模式，则Cache表现出来的优化效果良好，使得程序有充分时间去重排、优化这些IO数据从而为更高效的并行写入磁盘做准备。对于随机大块写IO，此时Cache效率最为低下，整体瓶颈归于磁盘但是表于Cache，因为Cache快速被充满而后端磁盘处于瓶颈态，此时系统最为难受。

增加Cache会对连续大块IO性能提升比较明显，但是对于小块IO，Cache空间足够，再增加也无济于事。不过依然要综合来看，比如如果太多客户端写入导致Cache不够用，那当然就考虑增加Cache，总之，都是相对的。只要后端磁盘没有达到瓶颈，增加Cache有好处，而且大块和小块，在Cache相对很大的时候，大和小的区别就可以忽略了。

不知道其他厂商产品底层如何flush的，我猜测应该是一直在flush，也就是源源不断地利用满后端的带宽。如果一次flush非要等待所有dirty block写盘，而此时其他线程都挂死的话，那绝对是划不来的。所以我推测是少食多餐制，flush频率非常快，或者是完全异步处理，后端细水

长流，而前端洪水暴发进入水库蓄水，同时进行。而某些产品受制于日志方式，日志保存的地方太小以至于flush频繁并且是同步阻塞模式。前者那种细水长流模式下，Cache越大当然越好，再大不怕因为电池保护；而少食多餐模式下，太大的Cache对写IO也没什么效果。而后者那种电池只保护日志空间的模式下，则太大的Cache没有用处，充其量读缓存占大部分，剩余的写缓存根据保存日志的空间来定。

上述是Cache管理的大概思想，至于更多细节的实现方式，比如多流合并、分区等思想和技术，就要看不同产品不同设计方式和开发人员富有创造力的头脑了。

但我有个阵列，把Cache由原来的1GB增加到3GB。做SPC1测试从BSU 240增加到300，增加了25%，是因为原来Cache就太少了么？

不否认这个测试结果，但是他的系统缓存只有1GB，未免太小，况且spc结果中随机写这部分到底提升了多少，无从考证，况且不知道是什么产品，底层destage的机制等。但是不可否认的是，增加Cache，对读优化肯定是增加的了。

更多的缓存意味着可以将更多的IO直接写到缓存里面，马上response给主机说IO complete，然后等积累到一定程度在后台destage给磁盘，性能应该更好才对，为何说是相对的呢？

Cache太大，有太多的数据没有destage，虽然放到Cache中有电池保护，但是还是要尽快destage到硬盘上才保险的，这就引出了我对于destage方式的讨论，如果是一次全部destage，那么太大的Cache当然会耗费很多资源了，所以上面才请林总介绍一下目前大Cache的产品destage是怎么个机制啊。

当然，后端通道和磁盘越多，相应Cache就得跟上去，这个无可否认。而且如果他的destage方式是按照Cache容量比例来的，而且后端没有瓶颈，那当然是缓冲空间越大越好了，如果后端捉襟见肘，还配百八十GB的缓存，那也是没必要的。但是为何我说随机IO下Cache优化效果最小，也就是性能提升最小，因为毕竟是随机IO，缓存再多，形成整条写的几率也是相对其他类型的IO低，这点必须承认，所以后端依然是会产生瓶颈的。既然这样，后端产生瓶颈的几率增大，那么必然需要destage相对频繁一些而不是积攒太多一次destage，因为一旦积攒太多而待写数据过于离散，则后端需要很长时间才能完成destage，资源消耗比较大；反过来说，destage如果频繁，则Cache对于写数据所占的空间也就不需要太大了，而可以放更多的读prefetch数据。这方面可以继续讨论研究。

57. SAS和FC

据说SAS接口的普通盘阵，十几块盘就可以达到**600MB/s**的吞吐量，这样我还用买高端FC盘阵么？

SAS盘阵达到700MB/s，800MB/s，900MB/s的速度都是常有的事情。SAS卡上的多路SAS PHY可以形成宽端口的，一般可以达到12Gb的带宽，也就是4个3Gb的合并，所以才能有这个速度。目前SAS已经可以达到单PHY速率6Gb/s。SAS相对FC是不错的选择。至于高端的FC盘阵，往往其他功能强，接口也多，而且线缆方面也比SAS方便，更容易扩大至很大的扩展柜数量，所以SAS目前并未非常流行，取代FC的路程还有很长要走，让我们拭目以待。

我一直有个疑问，就是SAS在Sequential环境中的表现到底如何？理论上可是有**12Gb**带宽啊。从我看到的几个产品上看，好像是用SAS

的产品在**Seq**环境中带宽表现不是十分好，反而在**random**环境效果更佳。

不知道是哪款产品，怎么测的，**SAS—RAID**卡市面上有很多：软，硬，半软半硬。带**RAID**功能的**SAS**箱的表现与外置**RAID**卡再接**JBOD**的性能没什么差别。随机和**seq**环境下的性能主要取决于参差不齐的**RAID**卡了，看其优化是否到位。能否提供一下实际数据，研究研究。

58. 软**RAID**和**RAID**控制器有何不同？

基于软件的**RAID**和基于控制器的**RAID**有什么区别？

前者泛指利用主机自己的硬件资源来实现**RAID**。由于主机上没有针对**RAID**的硬件芯片比如**XOR**运算器或者其他硬逻辑芯片，所有**RAID**功能全部由主机上的**CPU**通过运行操作系统底层的**RAID**逻辑代码执行。后者泛指**RAID**逻辑运行在独立的硬件上，这些独立设备一般都配有特殊的硬**ASIC**芯片来将复杂的运算硬化入芯片从而相对于全靠**CPU**来运行的系统性能大为提升。

当然，也有的控制器全部用**CPU**来执行**RAID**功能和其他存储系统的所有功能。二者的区别：前者耗费主机资源，不过如果主机**CPU**不繁忙、系统负载不高而且**CPU**内存够强劲，其表现的性能不亚于硬**RAID**。后者不耗费主机资源，但是要求一个独立的设备，成本增高；前者由于在主机上运行，一般不考虑实现复杂的逻辑而尽量保持高效率，后者则由于在独立的设备上，可以方便实现更多的复杂功能；前者运行于主机之上，一旦出现问题或者需要维护，则很可能要停机，而后者如果不出现大问题，对主机端一般没多少影响，灵活。

59. 关于RAID的初始化

几块盘创建好一个**RAID 5**后，一般是采用什么方式初始化的？我现在想到有两种：

(1) 每个**Stripe**都是写**0**，这样只需要进行写操作，不需要进行读操作和**XOR**操作，而且最后的**Stripe**数据是呈异或关系的。

(2) 每个**Stripe**是模拟**RAID 5**坏掉一块盘后插入一块新盘重新**recovery**，即先读同步的盘上的数据然后作**XOR**，计算结果最后写入被同步的盘。

我知道**Linux**内核中是采用后一种方式的，但是在性能较弱的**CPU**上会显得有点吃力，因为要作**XOR**计算，不知道有没有其他**RAID**厂家采用第一种方式的？

另外，不知道有没有不做初始化就直接使用的**RAID**厂家？

前台初始化的时候是不能接受IO的，必须等待。比如**DS 6800**就是这样。前台初始化理论上也有两种方式，也就是上面列出来的两种方式，**Zero Disk**，或者读出整个条带的原来数据然后计算**Parity**数据然后写**Parity**。不过一般倾向于**Zero Disk**，因为这种方式最为简便，只需要发送一个**SCSI Operation 0x01**指令到磁盘然后等待返回即可。控制器不耗费一点资源。如果是第二种，则有点兴师动众的意思，速度慢效率低，也没有必要这样做。

后台初始化，这种方式可以接受IO，理论上是控制器从**RAID Group**被建好的那一刻起即开始做初始化，不可能使用**Zero Disk**的方法了，因为此时需要接受上层下发的IO了，必须由控制器做读写操作，所

以只能用Parity重算的方法。一旦某个Stripe有IO进来，则控制器跳转到这个Stripe，初始化这个Stripe，捎带上下发进来的写数据一起将这个Stripe处理好，使得其Parity与user Data是一致的。如果是读IO，而且目标地址落在了尚未初始化的条带上，那么控制器直接在内存中返回全0给发起IO的主机；如果落在了已经初始化过的条带中，则此时由于控制器无法判断这个条带初始化之后是否已经被写入实际数据，只能假设已经被更改过，所以需要从对应目标地址读出数据并返回。处理完后，继续跳回原来的断点然后继续顺序处理其他Stripe。控制器在后台初始化过程中会对已经初始化和未初始化的条带做记录，这个记录很小，比如一个bitmap，每个位代表一个条带，初始化完成就标为1，未初始化就标为0，所以查找起来不费力。

为了避免不必要的麻烦，前台初始化过程应当选择Zero Disk的方式。校验型RAID Group必须初始化，只不过是前台还是后台而已。有后台初始化的产品，一些低端RAID卡都可以做到。

前提是RAID 5边初始化，边要写数据。当RAID 5初始化到了10%，而数据写到了20%。假如这时RAID 5中掉了一个盘，请问我这写的20%数据能够恢复吗？如果不能，前10%的数据能恢复吗？或者数据就是不能恢复？能举个例子帮分析一下吗？谢谢。

RAID 5本身是容许一个磁盘损坏或者丢失。不管前台或者后台初始化，凡是已经初始化完成的Stripe，皆可以容许一块磁盘丢失，而未初始化完成的Stripe，保护它们也没有意义。这样的话，系统必须依然可以从剩余的磁盘来校验出丢失磁盘对应Stripe上的数据。不过实际上，一个未初始化完成的RAID Group已经坏盘，那么正好是破罐子破摔了。你想想，此时系统就等于初始化完成了并且进入等待Rebuild状态，插入好盘之后，重新逐Stripe的Rebuild，此时就无所谓初始化与否了。

所以，不知道上面这个例子是假想的还是实际的，理论上讲系统依然可以接受IO，只不过一旦未被初始化的Stripe上有IO，则数据一致性要靠应用层来判断了。所以上面这个例子应该不会发生，如果实际中真的发生了RAID Group因为未初始化完而掉盘就停止接受IO，则可能系统为了充分保证安全性吧，此时你可以插上新盘让它Rebuild就可以了。如果这样都不行，那只能证明这个后台初始化系统设计的不完善。数据是可以手动恢复的，找专门的恢复结构，只要他用的RAID 5校验方式是常规套路。

60. 关于SAN共享

SAN 为什么不能文件共享？谢谢！也就是说，为什么块级存储，不能提供共享？

SAN是一个大家访问存储设备的通道，处于底层。对于一个目标设备，可以有多个客户端设备对其访问，同时读写，没有问题。块级的访问是最低级别的，接受访问的设备根本就不去管有多少客户端访问，谁写了哪些扇区，是否被不正确地覆盖了。

底层就是要实现高效率的访问，如果把这些东西都做到硬盘上或者SAN盘阵中，那逻辑就相当复杂了。当然，NAS设备就是这样一个提供文件共享的设备，因为NAS协议的一个作用就是解决多客户端共享文件，保证文件的一致性。然而这也取决于客户端的决定，如果客户端在使用NAS的时候，不进行Lock操作，那同样也会出现后来写入的覆盖之前写入的情况，就是这样设计的。这些逻辑全处于应用层上。

至于Block级别的Lock，SCSI Reservation算一个，但是它不会像NAS协议的Lock一样去保证一致性，其功能也不是为了共享，最大一个

作用是为了解决Brain Split。还是那句话，Block级别是低级智能的。要实现文件共享，必须在Block卷这一层解决，也就是使用集群文件系统，或者使用一个诸如Sanergy的卷共享软件，上层依然使用诸如NTFS这种常规的FS，卷共享软件可以保证访问同一LUN的多个客户端上的FS时刻从这个LUN读取的数据都是最新的数据，当某个客户端写入数据的时候，卷共享软件会将其他客户端FS缓存中对应被写入块的数据进行过期作废处理，重新读入最新数据。

各个客户端上的fs缓存，此时在卷共享软件的全盘管理下，实现了全局共控。如果不这么做，举个例子，假如T1时刻设备1将10000这个数值写入了扇区0，而设备2不知道，没有人通知它，设备2的FS缓存中对应这个扇区的数据依然是0。随后，设备2上的程序做了一次运算，比如：余额+5000，其实此时，余额应该=10000，但是由于没人通知设备2扇区0的内容已经更改，所以设备2仍然用0来当余额，这样算出的结果是5000，然后写入磁盘，之后，余额变成了5k，你赔了10000元啊！

上面的例子是一个概要，现实中，多设备同时读写同一LUN，会产生各种各样的后果。有时候设备1下次读出的时候会完全混乱，就像刚才的例子，应该是1 50000，结果错误保存成了5000。这种结果算是杀人于无形之间。有时候如果遇到FS的MetaData扇区不一致，则FS轻则fsck，重则崩溃。崩溃了比上一种情况好，如果不崩溃就这么以讹传讹将牛头不对马嘴的数据返回给上层，那后果可是不堪想象了。然而，用了上面的方法，只能保证数据在文件系统和卷之下逻辑的一致性，却不能保证上层业务方面的一致性。业务层的逻辑不可能也没有必要放到FS之下实现。所以还需要实现业务层的一致性，举个例子：比如设备1在这个LUN的0扇区写入了数据，而设备2在卷共享软件或者集群FS的控制下，它的FS缓存对应的这个扇区的数据过期，重新读入最新的由设备

1写入的数据，读入之后，设备2对其做了更改，写入覆盖扇区0。

这个过程是一个正常的过程，不会产生数据不一致性。然而，如果设备1和设备2之间没有足够的配合，比如，我用同一个账户在设备1和设备2同时登录进行数据更改，假设还是刚才那个银行账户吧，原来余额是0，T1时刻我在设备1上存入10000，设备1将数据写到扇区1保存，然后关闭终端应用程序。

此刻设备2缓存过期重新读入新扇区0数据。然后我来到设备2查看我的余额，仍是0，为什么呢？为何FS缓存都更新了但是终端显示仍然是呢？这是应用程序编写的时候没有考虑数据业务层的一致性，如果程序随时参考FS缓存内的最新数据而不是自身缓存内的数据，那么最新的数据就会生效，不管是使用Push还是Pull的方式获得最新数据。T2时刻，在设备2上直接退出应用程序，此时设备2上的应用程序由于这个错误，将一直缓存在程序Buffer中的数值0写入扇区0，这样，10000变成了0。你损失了10000。所以，业务层面，也就是应用程序层面，一定要考虑周到，利用各种方式来相互通信，或者从底层时刻获得最新的数据，是很重要的。

61. Cache和Buffer的区别

请问，**Cache**和**Buffer**的主要区别是什么？都是缓存，区别在哪？

其实Cache和Buffer，物理上讲都是RAM。逻辑上讲，你把Cache叫成Buffer，或者把Buffer叫成Cache，都没有错。不过Buffer多用于编程方面，Cache多用于非编程方面的叫法。比如为某程序分配一段Buffer，而一般没有说为某程序分配一段Cache的，但是你可以说这个程序有Cache，或者说Cache是泛指，Buffer是特指。见仁见智。而对于磁盘阵

列来讲， $\text{Buffer} = \text{Cache}$ 。

另外，从本质上讲，**Buffer**是“缓冲”，而**Cache**是“缓存”，即**Buffer**中的数据是一定要在短时间内被处理的，而**Cache**则可以作为一个数据的长期的容器而其中的数据不一定非要被立刻处理。

62. 关于存储系统中的**CPU**的作用

高端存储中**CPU**起的作用到底有多大？

高端存储的**CPU**对性能影响很大。因为高端产品其出发点不单单是把磁盘数据拿出来扔出去，或者等着别人写进来，然后写到磁盘这么简单了。高端存储中**CPU**主要决定下列4个大方面的作用：

（1）基本的数据吞吐服务。

（2）物理上，大量的IO卡和接口，需要消除底层的瓶颈。总控各种附加硬asic，比如xor的芯片，或者数据压缩硬芯片等功能芯片，使**CPU**周边的所有枪杆子和部队有条不紊地执行任务。

（3）逻辑上，大量客户端并发、随机IO操作的优化，需要消除逻辑上IO的瓶颈。

（4）功能的多样化，各种高附加值的功能比如snapshot、mirror、dr、dedup等。

CPU的高性能对整个存储的性能起到多大的作用？

有了上面的4个方面，下面就来一一描述一下**CPU**如何影响其性能。

（1）基本的数据吞吐服务。

这是一个盘阵最基本的功能。prefetch、queue、read write、flush，这些过程都是最基本的要求。这些过程，看算法复杂度和IO类型而定，基本上连续大块IO对CPU耗费不大，主要在于磁盘和Cache的瓶颈，因为连续大块IO不要求CPU做出多少运算。而对于连续小块IO，此时虽然IO是连续的，但是Size变小，系统整体吞吐量相对于大块IO来说降低不多，当然，前提是除了磁盘之外的其他节点没有瓶颈，而CPU利用率却显著上升，此时对CPU的要求就逐渐显现出来。

当前端的IO逐渐增加，CPU—Cache—diskchannel这条线上随处可以产生瓶颈，比较好的表现应该是disk首先瓶颈，如果是Cache或者CPU首先瓶颈，那么这个系统就不是最优的。对于随机IO来讲，此时程序会进入优化随机IO的算法模块中，视算法复杂度而定，此时要求CPU足够强劲来抵消一部分磁盘固有的面对随机IO的瓶颈。此时，算法越精良，CPU越强，性能就越提升，当然如果算法本身已经达到瓶颈，此时提升CPU也没有用。所以，这些东西都要经过详细的测试、考察。

（2）物理上，大量的IO卡和接口，需要消除底层的瓶颈。总控各种附加硬asic，比如xor的芯片，或者数据压缩硬芯片等功能芯片，使CPU周边的所有枪杆子和部队有条不紊地执行任务。高端存储有大量的IO卡设备，从底层角度来讲，如此多的IO设备和接口，就要求多CPU和够多的总线与其对应来响应源源不断的中断和数据收发操作了。其次，高端存储的架构大多硬件模块化，各个模块细分功能，比如xor模块专门计算xor值，其他模块fc通道控制、Cache控制、数据压缩等，这些硬件芯片在一个高端存储中有多个，这些功能的相互配合和运作要求CPU数量足够与之匹配，而频率则没有过多要求。

(3) 逻辑上，大量客户端并发、随机IO操作的优化，需要消除逻辑上IO的瓶颈。

高端存储的一个必须考虑的东西就是同时满足大量客户端的并发操作。大量的不同种类的IO类型同时进入，此时要求算法能够临危不惧不乱，井井有条地对这些IO进行queue、requeue分类等操作。首先算法本身应该效率足够高，其次CPU应当足够强或者核心足够多从而使得算法更快地执行而不产生等待，这样才能满足大量数据源源不断地进出而不是堵塞在Cache中，高端存储Cache动辄几百GB，这就要求操作手需要更快地充满或者清空这些空间以便接受更多IO。

(4) 功能的多样化，各种高附加值的功能比如Snapshot、Mirror、DR、Dedup、Cache分区等。这些东西可以说是纯软件操作了，对磁盘速度没有过高要求。而对算法要求很高，比如Snapshot、Dedup等，你看Datadomain为什么两家来抢他，就是因为它的算法能让Dedup在在线数据上运行而效率足够高。越是好的算法，越是能在耗费CPU相对较小的情况下完成相对较好的任务，如果算法不好，CPU利用率又高，那只能提升CPU来补偿了。还有诸如Snapshot、Mirror、Sync等操作，其底层是很复杂的东西，其底层要保存很多结构比如bitmap之类，都要求算法和CPU的。

个人认为高端存储主要在于磁盘数据与缓存的交换，那么主要性能就体现在这里，那**CPU**的性能体现在哪里？

刚才已经列举了高端存储除了基本的数据吞吐服务之外的功能以及CPU对其影响，这里就不多说了。

仅仅是对数据预读或控制读写队列？那么现有的**CPU**是不是足以

满足需要？

如果仅仅是prefetch、queue optimization，RAID卡上的CPU也做的不错，但是高端存储需要更多的prefetch，更多更复杂的queue，更多更复杂的算法，考虑的更多，所以需要CPU足够强劲。现有的CPU都是根据整个系统可以提供的动力来选用的，或者存在商业价值的因素，不满足要求就花钱买更强的CPU。

HDS、EMC的高端中采用的类似分布式的结构，采用大量相对低频率的**CPU**；而**IBM**则是对称多处理器结构，抛开可靠性等因素单纯考虑控制器性能，**IBM**的**DS8000**的结构通过提升小机性能来提升存储性能，究竟能提升到什么程度？

CPU性能提升有两种方式：整体核心数和整体频率和。这两种提升方式的选择，与系统软件底层结构有很大关系。如果系统底层的各个模块之间是互不牵制，独立并发运行的多个进程或者线程，并且明显存在线程并发数已经受到整体CPU核数的限制成为瓶颈了，那么这种系统采用提升CPU核心数量的方法，性能提升最为有效。

而如果某系统软件底层采用的多是各个模块之前有牵制不能并发运行，或者直接单线程，这种结构很容易受到CPU频率的限制，而提升核心数对这种结构没有很大的提升，反而提升单个CPU的频率，性能提升很大。然而，不管什么样的软件架构，随着产生瓶颈的触发因素不同，比如IO的行为、并发量或者其他功能性模块比如snapshot等设计不同，对这两种架构产生的影响也不同，有时候前者反而可能受频率影响，而后者可能受核心数的影响，这时候就需要综合判断取平衡了。**DS8000**软件层面采用什么方式，我不清楚。但是无外乎上面的两种架构。

IBM把P6装到DS8000里能起多大作用？

这不是起多大作用的问题，而是这个产品就是这么设计的，用小机充当控制器。

63. 数据库系统与文件系统的关系

数据库系统与文件系统的区别是什么？最近买了《大话存储》，正在读，感觉不错。

数据库是应用程序，一般都是运行在操作系统之上的。而文件系统是操作系统内核的一个模块，运行于操作系统内核。然而，数据库也有文件系统的功能，即，可以自己管理和分配磁盘上的Block，读写数据，当然也可以利用OS内核的文件系统来读写文件数据。

64. Iometer中Worker的问题

您好，我想请教您一下在使用IOmeter进行测试的时候，多个worker连一个盘和一个worker连一个盘测试有什么区别呢？谢谢！还有当经过samba和NAS导出的盘显示成黄色红杠的时候，怎么开始测试。IOmeter写入iobw.tst文件的时候，怎么写入，写入多大啊？非常感谢您！

如果你这多个盘是同一台盘阵，如果想测试整体性能，得需要知道这个盘阵内部是如何对IO进行处理的，如果只测试一个盘，可能并不能反映这台设备的真实能力，因为设备有可能基于LUN来分配资源，如果只对一个LUN进行IO测试，则这样的话很有可能盘阵列就有所保留，所以最好是每个盘都多个worker测试。Iometer好像不能对NAS盘符的，可

以对iSCSI。那个文件是要充满整个盘的，如果直接测试raw设备，则不会写文件。测试格式化后的磁盘，会一直写满。这个不用我说你自己试试也能试出来。

65. 测试用Block大小的问题

网络存储测试中测试存储系统性能的IOPS时，为什么块大小选择512B~64KB啊？512B肯定是因为磁盘的扇区原因，可64KB是怎么回事？

要获取IOPS，IO Size当然要最小，也就是512B了，IO Size过大的话，会在IOPS较低的情况下就已经达到饱和链路带宽了。此时不足以反映设备的真实饱和IOPS。达到最大IOPS的条件是完全利用所有前端接口发送请求，512B的连续IO，并且链路带宽没有饱和，此时所得到的IOPS数据便是系统饱和IOPS。

当然这种情况下的IOPS吞吐能力是没多少意义的。在小块随机IO情况下的IOPS更具有意义。更有甚者要求不使用内部Cache，甚至连物理磁盘的读Cache都不使用，在这种情况下所得到的IOPS数据则可以反映设备对后端磁盘IO的优化程度。至于64KB，这个就因人而异了，总之，要根据环境来进行测试。比如你的环境中主机的IO Size，最好能做成一个图，取几个点，然后针对存储IOPS吞吐，结果再做一个图，这样比较直观地反映存储在各种Size和随机率IOPS情况，并判断这个设备是否比较均衡而不是大起大落。

但是我想测试网络存储系统的性能，也就是通过网络连接后的存储系统，希望能尽可能地模拟真实的环境，就需要不同的IO Size了。但是IO Size的选择有个范围，测试时想采用IO Size线性递增的方式进

行，即**512B**，**1KB**，**2KB**，**4KB**，..... **64KB**，.....，但是最大多大合理，是否是一直递增，直到系统的响应时间使用户不能接受为止。我在某个资料上看到**TCP**的窗口大小是**64KB**，不知道是不是跟这些因素有关？

根据你环境中主机的行为而定，如果主机平均在**64KB**，你去用**512B**测试出来看性能，那就不对口了。**IO Size**最大值是有的，两个地方限制：协议本身，设备本身。前者是协议固有限制，后者则根据存储端的考虑，有些存储设备并不严格遵循协议，这也是兼容性参差不齐的原因。最后你又说**TCP**了，那么我更不明白了，你这是测试的**iSCSI**还是**NAS**呢？**IO Size**和**TCP**一层的没有直接关系。**TCP**的**Buffer**大或者小，并不制约上层的东西，倒是**MSS**和**MTU**有点制约关系。

66. 存储性能的衡量指标

存储系统的性能衡量指标有哪些？目前，我用三个海量存储系统测试的性能指标（元数据吞吐率，并发访问量和聚合带宽**IOPS**/响应时间）来判断网络存储系统性能是否正确？由于网络存储系统测试时是通过文件系统接口来测量的，因此不知能否测出网络存**IOPS**/响应时间指标。在衡量磁盘阵列时，这个指标是必需的，但是哪个测试是基于文件系统之下得出来的？疑问中.....

元数据吞吐量，其实没有多少实量，看元数据**IOPS**。而对于**NAS**来说，这些**IO**大部分会**Cache hit**，因为**NAS**内部可以感知自己的**FS**逻辑，**FS**预读元数据。并发访问，聚合带宽，这两点**IOPS**响应时间曲线可以反映出一台设备大概的素质了。当然可以不通过文件系统接口，使用**RAW**直接读写测试。

可是在小文件模式下，元数据的量就上来了。

我的意思是说实量和虚量的概念。实量直接导致吞吐量上升，而虚量直接导致IOPS的上升，小文件多的情况下，虚量相对比较多，IOPS相对较大。

请教冬瓜兄，如果别人只给我们提供文件系统接口，我们只能把别人给的网络存储系统当成黑盒子，那我怎么得到系统的**IOPS**与其对应的响应时间？我知道在磁盘阵列里是阵列管理器决定了**IOPS**的大小，但是在存储系统中，由于磁盘阵列之上可能要做虚拟化和并行文件系统等，用**IOPS**与其对应的响应时间来衡量这样的存储系统可信么？还有几个问题：看元数据更应该看**IOPS**？元数据操作速率和**IOPS**的关系怎样？

看来你的存储空间就是一个基于NAS的Volume。如果你使用linux/unix系统，那么可以将NAS客户端也就是主机端上的NAS缓存关闭，mount选项中有好几项可以控制这个行为，这样得出来的IO是可以排除本地缓存加速的裸IO结果。不管怎么虚拟化，怎么并行，最终我们要的是应用程序能够得到快速的IO处理服务，而IO测试软件就是可以从一定角度反映应用程序获得这种服务的具体情况的。**IOPS**和响应时间，以及吞吐量，这是最终我们需要看的，当然可信。所谓元数据操作，在NAS中，比如NFS中的getattr()、fsinfo()、fsstat()、lookup()、create()等。这些操作根本没有实量，也就是IO中没有传输实际文件数据，而都是元数据，这个你去抓包看看就理解了。这些没有实量的IO，块小，频繁，容易导致**IOPS**上升而吞吐量很低。

67. 单块硬盘的**IOPS**问题

一块硬盘的**IOPS**约为多少？通常看储存的性能**IOPS**和**Throughput**这两个参数，那一块**300GB SAS**硬盘的**IOPS**大约多少？有谁知道，单块硬盘，不配置**RAID**，不使用**RAID**卡的**IOPS**？

这个不能顺嘴就说。你可以自我测试一下。硬盘有Cache的，读**IOPS**在一定条件下甚至可以达到上千。小块随机**IOPS**，就很低了，几十都有可能。通常情况下，综合来说，**15000 RPM**的**FC**和**SAS**能有**300**差不多，**SATA**，差不多**160**吧，这是**8 / 2**比例混合读写时的大概结果。如果是**100%**随机读，那么大概**400**左右；**100%**比例随机写，不到**400**。

68. 何谓“端到端”？

端到端到底是什么意思？总看有些资料说“端到端”的什么什么，这个词到底什么意思啊？跟什么相对呀？端到端到底有什么好处？是支持的传输距离更远么？

端到端不是好处和坏处的問題，只是一个事实而已。端到端就像是字面上理解的一样，一端到另一端，**through**，穿透性的。具体还要根据上下文来解释。比如**TCP**是端到端有状态协议，意味着**TCP**是在通信双方的两端各保存状态机，不管两端之间经过什么链路，用什么设备相连，数据包走的哪条路，**TCP**不管，**TCP**管的就是收到对应的数据从而去触发状态机改变，依次循环下去。再比如描述存储设备，诸如“端到端**4Gb**带宽”，这里的端到端，就是说从存储前端到后端，接口带宽都是**4Gb**，而不是前端**4Gb**，后端降低到**2Gb**，这样后端就可能产生瓶颈。也许这就是所谓的“好处”吧，还是那句话，端到端和好处坏处没有关系，与传输距离更是没有一点关系。是一个事实而已。

69. **MSCS**对**iSCSI**阵列的要求

MSCS双机，使用**iSCSI**阵列，目前担心有些**iSCSI**阵列实现不是很完整，咨询一下**MSCS**对**iSCSI**阵列有什么特殊要求么？

只要注意**iSCSI**阵列支持**SCSI2 Reservation**即可，最好支持**SCSI3 Reservation**。**SCSI3**的**Reservation**是**SCSI2 Reservation**的进化版，专为并行多客户端访问**LUN**而生，**MSCS**是一种**HA**，而不是并行访问。**MSCS**用的是**SCSI2 Reservation**，也就是传统的**Reserve**和**Release**的command。**Break reservation**是要由客户端也就是主机端来实现，主机不**Release**，**Reserve**永远存在，**SCSI2**时候只要存储端的**SCSI Stack**重新**reset**或者设备断电重启后，即**Release**。**SCSI3**中简单的**reset**或者**reboot**并不能**release**。

70. IOPS与带宽的关系

IOPS与带宽有什么区别？存储阵列以哪个参数为性能指标？

（1）**IOPS**与带宽有什么区别？每次**IO**就是一次操作，比如读从哪开始的多少扇区的数据，这就是一次**IO**。每次**IO**请求的数据量乘以**IOPS**，就等于带宽。

（2）存储阵列以哪个参数为性能指标看客户端对存储的需求了，有些要求高带宽比如视频编辑，有些则要求高**IOPS**吞吐能力比如一些小块离散的**IO**。前者虽然达到高带宽但**IOPS**可能较低，因为每个**IO**请求的数据量很大，后者**IOPS**高但是可能带宽较低，因为每次**IO**请求的数据量小。

71. 随机与连续的比较

（1）随机读是不是比顺序读速度快？

你搞反了吧。随机读是不可能比连续读快的。这里“顺序”应为连续，顺序和并发是一对，连续和随机是一对。如果没有Cache的作用，随机读更是慢上加慢。有了Cache，有了prefetch预读的效果，能轻微增加随机读的效果，但是效果的提升不像连续读那么显著。另外，如果你用SSD，那么又另当别论了。SSD下随机IO和连续IO的差别没有机械硬盘那么大。

（2）随机写是不是比顺序写慢？

随机写同样比顺序写慢，如果没有Cache，那后果是很严重的。有了Cache，能大大提升随机写的速度，但是仍然比连续写要慢一些。即便有了Cache，还要看Cache策略，是write back模式还是write through模式，write through模式的话，随机写虽然比没有Cache情况下要快一些，但是依然很慢。write back模式下，只要Cache足够大，控制器将Data收进来并作妥善安置后便会对主机返回ack，IO完成，然后控制器会对这些随机的IO进行重新优化写盘。

（3）存储阵列的随机读写是不是比顺序读写性能差？

只要是随机，总比连续慢。不管单盘还是阵列，有无Cache。但是读和写之间谁快谁慢，就得看Cache了。

72. 关于ZONE和争抢资源的问题

“在设计Zone时，有一个最基本的原则——每个Zone中只有一台主机。遵循这样规则设计出的Zone，结构非常清晰，不会有错误产生。另外可以防止不同的主机争夺对磁盘控制权的情况。”争夺磁盘控

制权，如何理解？

那句话估计是作者估摸着乱说的，说的时候又没说底层机制，可以不必在意。并不存在所谓争抢问题，争抢是应该的，如果强者胜出，那岂不是只有一个Initiator可以访问target了么？这个所谓“争抢”是正常现象。所以说这个作者在估摸着乱说，只是为了体现Zone的重要性而已。SCSI3 PR用于并行访问，集群脑裂后的vote，与这个“争抢”就没有关系了，vote就是在争抢，有什么反常么？

73. LAN Free的问题

IP SAN 能做到LAN Free吗？如果是iSCSI必然用到Lan，那么还能Lan Free么？

我们用的是SL 500磁带库，不知道如何实现IP SAN环境下的 LAN Free？

呵呵，Lan Free这个概念早该被干掉了。取而代之的是Front-End Free。所谓Lan不一定就非得指代前端客户端与服务器通信的网络，你单独弄个交换机用于存储和主机，不一样是Lan Free么？但是这个单独交换机，是不是也是“Lan”啊？所以，这个词该被干掉了。用frontend和backend。后文对这些概念还有更加详细的论述。

74. 如何选择磁盘以及磁盘阵列的问题

冬瓜头，你好！很早就一直看你的文章，最近又重新拜读了你的《大话存储》。在存储应用中，我一直在思考几个问题：

（1）在数据库等关键应用环境下，一般建议配置146GB的

15krpm光纤磁盘，而不选择用大容量光纤磁盘。我的理解是：**146GB**的磁盘重构时间短，能够有效减少对应用的影响。但是在性能方面似乎没有提升。

（2）如何根据一个应用来选择相应的磁盘阵列（例如：磁盘阵列的前端端口数、后端磁盘通道数、**Cache**大小）？我的理解是：前端端口数=对应的**IOPS**/单块磁盘平均**IOPS**/主机端口平均并发磁盘数。但是**Cache**没有办法来计算了。主机平均并发磁盘数是否根据不同厂商不同，是否有经验值呢？

不知道，我的理解是否有偏差。还请帮助指正，谢谢！

感谢支持！就我的理解回答一下。

（1）你讲的有一定道理，即从**Rebuild**时间角度考虑。但是选择多大容量的硬盘，得根据盘位、价格、今后扩展、当前需求量等因素综合考虑。我想你说的“而不选择大容量**FC**磁盘”只是某种条件下做出的决定，没有普遍性。理论上讲，磁盘越多，性能越高，不管是从带宽吞吐量还是**IOPS**吞吐量的角度。如果对性能要求不高但是又要求大容量，那么没有理由不选择容量大的盘了。多一块盘，多耗电，多花成本。

（2）不同的应用的确会产生不同类型的**IO**，但是这不是用来选择盘阵的最重要的标准。如果你的盘阵只给一台主机、一个应用来用，那可以根据这个往下调查。但是多主机并行访问情况下，并行的**IO**在一起会将原本单路**IO**的属性混沌化，使得**IO**属性对盘阵的影响降为次要矛盾。此时主要看盘阵的硬件规格参数，还有其**SPC**评测理论值，**Cache IOPS**，非**Cache**的**IOPS**，即**Write Through**模式下的**IOPS**。首先需要弄清楚主机端到底可以以多大的速率、**IO**来轰炸存储，然后再根据存储的理

论规格来定，最直接的办法就是进行模拟测试，以实际数据为准。我不清楚“主机并发磁盘数”具体你指的是什么意思。

75. 所谓“坏扇区转移”功能

“另一个额外的容错功能是坏扇区转移（**Bad Sector Reassignment**）。坏扇区转移是当磁盘阵列系统发现磁盘有坏扇区时，以另一空白且无故障的扇区取代该扇区，以延长磁盘的使用寿命，减少坏磁盘的发生率以及系统的维护成本。所以坏扇区转移功能使磁盘阵列具有更好的容错性，同时使整个系统有最好的成本效益比。”上面这段话中，这个坏扇区转移是所有的商家都有这个技术，还是个别厂家的？

被忽悠了。这个技术是SCSI3中详细定义的协议，他只是实现罢了，没什么神秘的。去找份SCSI文档看看就知道了。

76. 关于不同设备不同IOPS的问题

向冬瓜头请教一个问题：**NETAPP**不同设备的**IOPS**数据的改变是依靠什么？软件结构还是硬件型号，例如**CPU**、主板、网卡还是什么？

CPU总线，外部接口，磁盘数量。**ONTAP**系统内核效率不同平台几乎相同的，发挥到多少就看硬件指标了。

77. 3块盘做RAID 6，坏2块怎么恢复？

3个盘做的**RAID 6**是如何实现掉2盘容错的？3盘做**RAID 6**的，

LSI8708可以做，那么怎么使用**1**块盘恢复其他**2**块盘呢？

不过，你就3块盘，有必要这样折腾？如果你明白**RAID 6**基本的算法，你就明白了。3块盘的**RAID 6**，我做个比喻吧，数据盘上的内容是中文，校验盘1上的数据是英文，校验盘2上的数据是德文。数据盘和校验盘1被拿掉，一样可以从德文盘经过翻译，回到中文。

本质上，**RAID 6**就是将数据盘用两种不同的校验算法校验出两套独立的校验数据来保存。对于三块盘的**RAID 6**，几乎所有算法算出来的其实等同于3盘**RAID 1**。

78. IOMETER测试问题

问题一是：我要用**Iometer**测试磁盘阵列的最大**IOPS**，**Irt**（**10%**负载下的响应时间）设置的参数里面有一项是并发**IO**请求数（在**Iometer**里面只能设置进程数，这个我知道），现在要确定这个参数的典型值（比如**10000**为基数，以**5000**为步长递增），但是我不知道如何确定这个值，我只知道**IOPS**的峰值**>800000**，我需要根据其他什么参数，如何来确定呢？我在网上看了很多测试的结果报告，发现大家并发数大多定为**64**，还有说是**128**或**256**的，这是为什么呢，如何确定的呢？用的**2**的几次方，这是什么原因？为什么不用整数比如我写的**10000**这样的呢？

这个并发**IO**请求数需要根据不同层次的限制来判断：**user level**和**kernal level**之间的限制，**device driver**处的限制，适配卡硬件**firmware**处的限制，存储设备处适配卡和**device driver**、**application**处的限制。取它们之间的最小值来作为这个并发**IO**数，再大的话，就已经任意一处被限制，没有意义了。至于每一处的数值，需要你去调查。这个数值也就是

queue depth，看到这个字眼就差不多是了。针对你的问题，10000是不可能的，也没必要，一般来讲，device driver和适配卡firmware处不会超过256的，64是折中数值。

问题二是：我看冬瓜头版主在别的帖子里回复说，一个系统测试**IOPS**、响应时间、吞吐量，这三个指标是基本的。我想请问，在随机**IO**应用环境下，需要测吞吐量吗？这个吞吐量是指什么？单位是**MB/s**吗？如果是的话，那在**Iometer**测试结果里面，不是有这个值么，还需要单独测试么？我的意思是在测试**lrt**和**IOPS**的时候就可以得出这个值。如果测量吞吐量的峰值，我知道应该是在顺序**IO**应用环境下测试对吧，问题是我现在在随机**IO**环境下，测吞吐量有意义吗？并且是可以直接通过**IOPS**块大小得出的啊。

这属于明知故问了。随机**IO**下主要看**IOPS**和响应时间，吞吐量意义不大。当然如果某个设备在相当随机和小块**IO**下依然能够饱和链路带宽，那是相当好的情况了。

79. 计算**IOPS**

假定：

- (1) 硬盘没有**Cache**，512B下随机**IOPS**为150
- (2) **RAID**卡和存储都没有**Cache**
- (3) 主机端**IO Size 512B**的随机**IO**，读写比例是2:3
- (4) **RAID 5**

问题，计算要**10000**的**IOPS**需要多少块盘？

$$10000 \times 0.4 + 10000 \times 0.6 \times 4 = 28000$$

$28000 \div 150 = 186.7$ 即需要**187**块盘才可以达到需求，

则最大的吞吐量为 **$10000 \times 512B = 5.12MB/s$**

请问，这样的计算是否正确？

你还少了一个假定，就是RAID控制器不能并发IO，也就是这个RAID 5时刻处于单个IO的读改写的状态。有了这些假设，就可以大致算出了。RAID 5写惩罚，在单个随机IO读改写模式下，共额外产生1个写和两个读。所以 $10000 \times 0.4 \times 2 + 10000 \times 0.6 \times 2 = 20000$ ， $20000 \div 150 = 134$ 。不过这个数字基本没多大意义。实际上，RAID 5视磁盘数量多少，可以并发不少IO的，最好的情况下是可以整条写，写惩罚没有读IO，而RAID 5任何情况下写惩罚中的写IO只有一个，就是写Parity。而随着并发几率增加，读改写或者重构写时的写惩罚更为严重，增加了大量的读操作。最后，还要看后端的总线或者Loop的限制，一个Loop所能提供的带宽和IO都有限。

80. 存储系统后端多链路的问题

存储设备后端接口数量有**2、4、8、16**等，如**EMC CX120**是**2**个，**CX240**是**4**个，**480**是**8**个等。可控制器都是双控的，后端有较多的链路除了起到冗余的作用，是否还有其他功能？本人理解一个控制器单位时间内只能使用一条链路，是否正解？

冗余是次要的，后端更多链路主要是为了接入更多磁盘啊！谁说控

制器单位时间只能使用一条链路的？看后端的芯片，一般位于同一共享总线上的多个链路是单位时间一条，但是高端货后端都是无阻塞的。

81. RAID类型与IOPS

情况是这样的，目前本人正在弄一家六百台的网吧，原先打算买两台游戏服务器（就是虚拟盘，把游戏放在服务器上面让客户机来读取，相当于网络存储了），原先是打算用四块**300G SAS**做**RAID 0**阵列。配上**32GB**内存做缓存（因为虚拟盘软件都有缓存功能，读过一次的东西会在缓存里，第二次读的时候直接读内存）。但这两天听了一个说法，讲的是**RAID 0**不能提高随机读写性能，所以没有必要做**RAID 0**，因为网吧读游戏都是随机的多。看了冬瓜头前辈的博客上有一篇文章，“深入理解各种**RAID**相对单盘速度的变化”，他的结论是**RAID 0**随机读写仅在并发**IO**+分割块很大的时候，**IOPS**显著增加。但对这句话不大理解，不知道在网吧这个案例中，到底有没有必要弄**RAID 0**。

不知道你网吧里什么架构，**RAID 0**不安全，坏一块盘数据全丢，除非找数据恢复公司，幸运的话能恢复一些。**RAID 0**是最好的提速方式，那个文章中我只是说相对于单盘在随机**IO**情况下的**IOPS**提升相对幅度。另外，需要完全深刻地了解**IO**类型，甚至存储系统，首先推荐你阅读《大话存储》一书。针对你的具体问题，说**RAID 0**不能提高随机读写性能，是错误的，当然可以提高，但是得看情况，有些条件显著提高，有些则不显著。

RAID 0随机读写，在并发**IO**+分割块大的情况下显著提升**IOPS**，你不理解这句话可能是因为分割块的问题，这里所谓分割块当时我没有用**Segment**这个词替代，如果替代了，你就明白了，如果还不明白，那

么我再说下去就要大费口舌了，就不如你去看看书来得快和益处多。到底用不用RAID 0，刚才也说了，RAID 0最好，但是你要忍受数据丢失的风险，综合评判，折中方案是使用RAID 5，既然客户端都是读而不写，RAID 5读的时候与RAID 0几乎相同效率。

82. IO冲突导致的性能下降

看到本版讨论的都是非常大型的存储，我都有点怕自己的小小系统问题拿不出手，不过还是鼓足点勇气请教大家了。不过我这个系统主要是我个人用的，对个人而言属于较为奢侈的存储设备了吧？由于本人的工作室内部资源分享需要，我需要组建一个六个客户端并发，每个客户端**50MB/s**的文件服务器，复制的都是较大的文件，一般超过**1GB**。（注意**50MB/s**是**M**字节每秒，不是比特每秒）。这意味着该服务器起码需要**300MB/s**的并发速度。为此，我的硬件配置为：

Dell 690工作站（**Intel 5110 cCPU / 4g ram / Intel 5000x**芯片 / **Intel 632x**南桥芯片组），**Intel 9402pt pci-e 4x** 双口网卡，共两块。**Highpoint 4320 pci-e 8x RAID**卡。**Netgear 724at** 支持 **802.3 ad**动态聚合的交换机。**4块Seagate es.2 500g**企业级**SATA**硬盘（备注，我本来打算要购买**8块WD RE3**的硬盘的，后来打算先随便测试看看能够跑多少，于是找了四块希捷硬盘来测）。我已经完成的部分测试：**Dell 690** 安装 **Win2008 SP2**正版系统，开启共享，关闭防火墙，四个硬盘通过**Highpoint** 做**RAID 0**测试。每个客户端都是**Win2008**系统，都是千兆**pci-e**网卡。

测试：

Intel 9402pt 两块，每块两端口，任何一个端口跟我的一台客户端

机（也是**Win2008 SP2**）连接，可以达到**90MB/s**的速度，这个时候只让这一个网口工作，不接交换机，不做网卡聚合。实际上就是单测一台客户端连接到**Dell 690**工作站上的速度。

测试：

将**Intel 9402pt** 两块总共四个端口做聚合**linkagg0**，**Netgear**交换机也做对应聚合。然后发动四个客户端通过**Netgear**交换机同时连接到**Dell 690**已经做好**RAID 0**的硬盘阵列复制东西，大约**50GB**数据，一个客户端**20MB/s**，一个客户端多于**10 MB/s**，另外一个**30MB/s**，另外一个也是**15MB/s**。

测试：

因为对上述结果迷惑不解，于是将网卡**linkaggr**去除，分散成独立的**4**个网卡端口，每个网卡端口直接接客户端机器（不经过交换机），这个时候的性能更加可怜，有的**40MB/s**，有的**5MB/s**，四个网口加起来总共速度才**75MB/s**。

测试：

Highpoint 4320卡为新购入，购入后我已经用它倒入**1000GB**的数据做稳定测试，没有任何问题。另外，四个**Seagate es.2 500g**硬盘组成的**RAID 0**阵列用**HDtune**测试速度在**250MB/s**左右。我反复做了很多次测试，但是就没有看到过四个网卡总共并发超过 **90MB/s**的情况，无论是网卡聚合还是网卡不聚合，无论是接交换机还是不接交换机。到此，我已经迷惑不解了，我虽然没有买专业化的存储，但是我购买的这些东西都是性能很好的东西，例如 **Intel 9402 pt**网卡，我放着好几块**8492mt**网卡没有用，花了将近**3000**元买了两块回来。例如**RAID**卡，我

没有用**Dell 690**的**RAID**，而是花了将近**4000**元购买了**Highpoint 4320**卡。另外交换机，都是正宗支持**802.3 ad**聚合的**Netgear 724at**交换机，一个交换机就是 **5000**元。至于网线，我全部是采用六类品牌线。所有这些部件单测都没有问题的。

我自己整理的怀疑思路：可以排除**CPU**、内存、交换机、网卡本身的问题，因为**CPU**源自始至终测试都是在**30%**以下，内存也足足有余。排除交换机是因为不接交换机，四个网卡口独自工作总带宽也上不去，网卡每个口我都做过几个小时大数据测试的。

剩下的怀疑：

（1）**Dell 690** 主板南桥有问题？只是猜测，南桥那么大的带宽，随便给点就超过**100MB/s**了。

（2）**Win2008 SMB**共享协议限制速度？或者这个协议并发能力上薄弱？

（3）**Seagate** 的 **SATA**硬盘阵列并发响应能力薄弱？虽然**SATA**的并发能力由于不支持**TCQ**，而只是支持**NCQ**，但是并发响应能力不至于如此薄弱吧？

瓶颈究竟出在什么地方？请有经验的高手帮帮我吧

这问题我遇到过，而且是在一台专业设备上，专业设备尚且如此，更不用说**Win2008**搭建起来的**DIY**了。总体来说还是软件效能问题，**CPU**内存和**IO**总线没有瓶颈，而且后端磁盘少说也得接近**200MB/s**才算可以（你读的大文件）。所以还是软件处的瓶颈。另外，交换机上的那种**aggr**最好别用，根据以前的实验，几乎无效。很大程度上与底层驱动

层面有关系，你如果换成FC的方式，则能够利用全部带宽，而不管你用CIFS还是ISCSI，得到的总是一块卡的效果，甚至还不到。可以换成WSS系统试一试，或者用FreeNAS，Openfiler，Opene等来DIY。

了解之前情况可以参看我前面的帖子。另外，我下周一还预约了测试更多的sas硬盘做成阵列，不过我几乎不抱希望了。

在之后又完成了如下测试。

一：服务器商那边测试。前日**Chinaunix**发完帖以后睡觉，睡到昨日下午。因为已经约好到一家专门做服务器的地方去看。这家服务器商既有品牌的味道，也有**DIY**的味道。见面寒暄过后，直接开测，平台为超微的**5500**平台，**RAID**卡为**RocketRAID 3560**，这个卡支持的硬盘比我的多，多达**24**个，但是我的性能比它的强，我的**IOP** 是**1.2GHz**的。用了四个 **ST es.2 1TB**企业级 **SATA**硬盘。结果性能跟我之前的测试差不多，三台机器并发测试就是**75MB/s**差不多。看来可以排除主板南桥的问题了。

二：回到家中换成**FreeNAS**把系统换成**FreeNAS**测试，居然并发性能比**Win2008**的差，看来**Win2008**的**SMB**性能比**Win2003**好不少，文件传输并发性能是不是比**Freebsd**好呢？有趣的是：我在三台客户端机器上同时从服务器上复制一个文件的时候，奇迹发生了，网卡居然可以跑到**900Mb/s**（我这时没有做交换机聚合，也没有做网卡聚合），也就是**110MB/s**，这个速度显然是非常令人惊讶的，千兆网卡的极限啊。我当时就开心的几乎跳起来了，然而我马上冷下来了，如果是不同的文件呢，于是立即开始测试，当三台机器每个机器从服务器上复制不同的文件的时候，整个速度立即下降，估计总共就是**45MB/s**，这个速度还不如**Win2008**下的呢。而且这个时候**SATA RAID 0**阵列速度

忽高忽低，一会儿是波峰**500MB/s**，一会波谷**60MB/s**，总体来说跳得非常厉害。于是得出结论，硬盘的并发能力是真正的瓶颈。另外，就是**Freebsd**的网络性能的确很强悍，我还从来没有见到过千兆网卡居然可以持续跑到**940MB/s**的。还有就是**UFS**文件系统性能是不是比较弱呢？怎么比**Win2008**的并发传输能力差啊。另外就是**Freebsd**的网络负载均衡能力强啊，几个客户端的速度非常平均，但是在**Win2008**下，高的**45MB/s**，低的**10MB/s**。

三：**15k6 SAS**硬盘上场。突然记起我还有一个**15k6 146G 15000转 SAS**硬盘，这是个好东西，于是立即接上。环境还是**FreeNAS**。三个客户端持续并发复制不同文件，速度总共是**18MB/s**左右，这个**SAS**传输东西的时候速度波动很少，基本上在**18MB/s**左右波动。这个跟宣传的持续读能力**110MB/s**差得太远了。但是一个客户端连接复制东西（其他客户端不动），速度可以轻松跑到**50MB/s**。

到目前为止的结论是：

（1）硬盘的并发性能是真正的瓶颈。我这几天看了些文章，没有确证但是我非常相信，人家**15个15k6 SAS**硬盘，持续大数据并发读写不过就是**100MB/s**多一点。看来一并发，硬盘阵列性能就下滑的非常厉害。

（2）网上的那些测试其实是针对单个客户端而言的，说白了就是机器的阵列就是机器自己用，如果组成阵列速度的确会很快。

（3）并发环境下的持续读写速度要达到**200MB/s**以上，估计得等到**SSD**硬盘了。靠目前的机械硬盘，即使是**SAS**，如果不像大公司那样，估计根本超不过**150MB/s**了。

(4) 我觉得现阶段要达到较高并发的办法（不一定对）就是在同一台机器里多组几个阵列，放不同的内容，让客户根据内容分流了。例如 5个15k6 SAS硬盘RAID，并发持续读能力也就是120MB/s，但是我如果分成5个盘，最好的情况下，五个客户分别读五个盘，那样的并发总性能就超过250MB/s了。但是这样一是无法像管理RAID那样一个整盘管理，另外一个方面对客户的需求无法良好统计从而优化分配。

(5) RAID阵列卡的作用。RAID阵列卡我觉得本身没有什么大的加速作用（这个跟厂商吹嘘的是两码事情，从这点可以看出，厂商是多么一致的忽悠我们最终客户），就是一点Cache，这个起不了什么大作用。但是我认为RAID阵列卡在保证数据安全方面还是比较有用的。另外，RAID阵列卡把多个硬盘的性能一定程度上串联起来了，这样针对应用而言，磁盘性能得到了很大提高。

今天凌晨完成的测试，就是在各位的提醒下做出的。测试目的：瓶颈究竟在哪里？测试服务器：Dell 690，测试硬盘：4块Seagate es.2 500g，1块 15k6 15000 转SAS硬盘，不做任何RAID，两块Intel 9402pt网卡四个口聚合，交换机做四个口聚合。Win2008 SP2系统测试客户端：4台笔记本电脑，1台PC，均是千兆网卡。测试方法：每台电脑分别对应服务器上的一个硬盘，这样避免硬盘的并发。测试结果：五台客户端并发的时候，Win2008 SP2任务管理器以及每个客户端分别监测，从服务器出来的流量稳定在2400Mbps，也就是300MB/s每秒。

测试结论：

(1) 网卡聚合和交换机聚合是成功的，网络上许多朋友说无法聚合，那是因为自己的交换机太差，不支持802.3 ad，例如H3C 1216 / h3c 1224。就我所知道的，5000元以下几乎没有支持802.3 ad的交换机

的，**CISCO**有支持的但是端口是**100Mb/s**的。

(2) 真正的瓶颈显然清楚了，那就是硬盘阵列。之前做的**RAID 0**阵列死活上不了**75MB/s**，就是因为并发性能很差，现在我把硬盘打散不做聚合，各自复制各自的东西，结果性能就上来了。

(3) 未来的希望要靠**SSD**了，同时我要高度鄙视一下硬盘厂商，鄙视**RAID**卡厂商，在很大程度上他们隐恶扬善，强调单一任务的性能，对并发性能只字不提。

另外，我个人花费了大量的时间和精力来测试，同时由于自己之前的一些无知，导致购买了不少价值比较高的设备，现在哭啊，大家安慰我一下吧。真希望**SSD**硬盘**1TB**降价到**3000**元以内，如果真的降价到这个程度，明天我就过去扛几块回来。下周一，我将做多个**SAS 15k6**硬盘的**RAID**测试，到时候再发上来。最后再次谢谢冬瓜头等诸位兄弟们。姐姐妹妹，估计这儿没有吧，如果有，一并感谢。

嗯，并发的大块连续读写效率确实低下的。读同一个文件那是**Cache**的速率了，所以能撑满带宽。并发情况下，而且数据是按照横向条带分布的话，由于磁盘寻道产生的致命影响，多盘获得的**IOPS**和吞吐量提升基本上相对单盘幅度很小。这是所有产品都不能避免的。**RAID 0**对单个客户端的大块连续读写提升还是很大的，这不能怪**RAID**卡，而是理论上已经不可能达到了，除非分多个**RAID**组，或者算法上提供用户自己决定数据分布的选项，但是低端产品别指望有这些，甚至高端产品灵活性也没有这么高的。楼主可以继续做如下实验。

实验1: **Size**适中的并发的随机**IO**。**RAID 0**对这个很拿手，与单盘对比效率定大增。

实验2：做两组RAID 0，两个盘一个，使其互不影响，调查数据并作对比。IO Size相比Stripe Size较大的时候，也就是楼主的环境，读写大文件，RAID 0就不支持并发IO的。其他RAID类型和IO类型，可以套用本书第4章中的几个列表。

不错不错，通过自己的试验有条有理地解决了疑问。不过，你说RAID对并发不行，也是偏激了。RAID 0如果增加条带深度，是可以并发成功的，你看那些分布式集群存储，比如XIV，等效条带深度为1MB，比如3PAR，等效条带深度为256MB，局部连续地址范围很大，所以这样的话系统整体依然可以并发多个上层IO，否则也不可能达到几万几十万的IOPS。至于一些小打小闹的RAID卡，其条带深度几百k而已，局部连续地址范围太小，导致一个上层IO同时占用了多个Segment，此时完全不能并发IO了，多个IO得排队，寻道，传输。

说实话，纯种RAID 3估计现在没有设备可以真正实现了，首先它要求的磁盘磁头位置同步以及转速的严格同步，这技术现代的存储系统都不考虑了，现在RAID 3都是假RAID 3，即条带很小的RAID 4，RAID 3—4—5的演变过程鄙人几年前有贴讨论过，《大话存储》中也有收录。RAID 3更不可能并发IO的，所以并发环境下，还是多组RAID，或者条带深度达到几百兆级别的高端虚拟化级别的设备，方能满足要求，此时由于算法的革新，SATA盘也就可以达到很好的整体性能。

83. MPIO和MCS的关系

您好，请教个问题，**MPIO**和**MCS**负载均衡，具体性能上会有差异吗？**MCS**是每个**session**多**connection**，与**MPIO**多个**session**具体区别在哪里？另外如果我的存储两种方式的负载均衡都支持，我可不可以先做**MPIO**的负载均衡，再对每个**session**做**MCS**的负载均衡？盼赐

教，谢谢！

MPIO里没有session的概念，MPIO没有连接状态，它不是一个通信协议。MPIO工作在SCSI层之上，iSCSI的MCS（Multi Connections per Session）（你问的时候应该注明是iSCSI，否则别人不知道你指什么，幸好我猜出来了）工作在TCP层，MPIO→SCSI→iSCSI/FCP layer4→TCP/FC lower layer→IP/FC lower layer→MAC/FC lower layer→physical bit encoder→physical cable，这是整体层次。iSCSI如果使用MCS，只是底层连接链路，或者TCP连接个数（使用同一个物理接口）上增多了，MCS层上发现的还是同一份LUN的一个影像。而MPIO层是在SCSI之上了，例如，在iSCSI Initiator处不使用MCS，通过两个网卡认到同一个LUN的两份影像，这时候就得用到MPIO层次的东西了。你问到的先用MCS对每个session做均衡，然后多个session再用MPIO来管理，当然可以。

84. 链路负载均衡的设计

我想做一个从服务器网卡到存储设备的网卡之间的线路冗余（包括网卡和线路交换机也冗余）的方案，每台服务器配三个网卡（一个做HA的心跳线，另两个分配同一网络的同一网段的IP地址，如A网卡是x.x.2.1，B网卡是x.x.2.2），把两个网卡的网线分别连到两个交换机上（就当它是个普通的二层千兆交换机）。存储设备是一个普通的iSCSI设备，有四个网络接口，两个交换机上分别连到存储设备的A接口和B接口。这个存储设备支持线路捆绑，可以把A和B网卡捆绑形成一个IP。不知道这样的方案能不能达到我所需的效果？服务器那边是否也要对两个网卡进行捆绑？交换机是否要求要三层交换机？望高人指点，谢谢！

(1) 存储端支持链路聚合，是什么协议？Etherchannel还是基于LACP的Etherchannel。但是又是用两条线分别连接两个交换机，这样的话，线路上没有什么特殊聚合协议，交换机端就不会管什么Channel了，不是端到端的聚合，而只是存储端一端自己的聚合，而且尚不知道存储端是否支持这种一头自己聚合。

(2) 存储端是否支持ISCSI的Multi Connection per Session，即MCS功能，如果支持，就多了一种选择，相当于可以负载均衡的多路径软件。如果不支持，就只能用ISCSI Initiator自带的MPIO功能。不过还是推荐MPIO。

(3) 存储端是否具有一种叫做IP Fast Path或者类似功能，即外出流量Bypass路由表，而根据进入的IP Packet所流经端口而直接转发出去，类似记录session。如果不支持，就不太好办了，必须将两台交换机级联起来形成一个广播域。

用不着三层交换机。如果存储端支持自己本身的聚合，则交换机最好连起来，因为你弄不清它如何选择外出转发链路的，除非你有把握。

(1) 存储设备支持以太网的链路聚合，从存储端用两条线分别连接两个交换机，是考虑到交换机的冗余。如果把捆绑后的两条线路都接到同一个交换机上的话，那么这交换机坏了，线路自然断了，我也不知道设备是否支持存储单端的链路聚合，所以不知道此方法是否可行？

(2) 存储设备应该是不支持MCS功能的，您说的在ISCSI Initiator使用MPIO功能，此功能在Microsoft ISCSI Initiator有吗？在什么地方？

(3) 存储端应该是不支持**Bypass**路由表的，我的这个拓扑中，服务器的两个网卡**IP**、存储设备的网卡**IP**都是同一网段的，这种情况下好像并不需要用到三层的路由表吧？

(4) 另外，至于您说的“如果存储端支持自己本身的聚合，则交换机最好连起来，因为你弄不清它如何选择外出转发链路的，除非你有把握”，你所指的交换机连起来是指交换级连吗？还是交换机之间互连，若这样的话就形成二层环路了。假如服务器的两网卡**IP**是同一段**IP**的话，就是不知道会用哪个**IP**跟存储通信？至于回流的数据（即从存储到服务器方向的数据流）的路径应该是可以确定的，从哪条路过来就从哪条路回去。

呵呵，怎么可能成环.....难道你要把服务器看做一个转发以太网广播的网桥？可以，两个卡你给桥接起来，成环了。你怎么能确定回流的它就哪条进来哪条回去？这个功能是要单列的，普通路由表模式下就是有一个默认的出口的。**session**模式下才会哪条进哪条出，那是一些路由器之类才有的功能，存储端当然也应该实现，你最好和厂家去确认一下。

85. 关于**D2D2T**

我们公司现在有一个这样的**FTP**应用，使用磁盘阵列以及磁带库进行二级存储，现在想在磁盘阵列上开辟一个区域用于缓存磁带库数据。这个缓存怎么做，有现成的算法或产品吗？

你的需求貌似是一个分级存储的需求。既然你的盘阵可以直连磁带库，那么就证明它已经开发了相关的模块，你要在它上面放缓存，我理解缓存中的数据是你近期可能要调用的所以不想马上就放磁带，那么你

得和这个盘阵的厂家联系看看是否有这个功能。软件方面灵活度就高了，一般D2D2T的备份软件都提供这种功能，只不过不一定叫“缓存”什么的，一般备份软件都把这种模式叫做Disk Stage，你可以选择什么时候将Disk中的数据转移到磁带。

86. 断开ISCSI Session时总是报错

为什么用Windows自带的ISCSI客户端去连接IP SAN，并且写入数据后，想在客户端把Target退掉，为什么老是会报错（提示session的问题），老是退不掉，是微软本身的问题吗？谢谢！

确保所有与本磁盘操作有关的程序都推出，比如打开的窗口。还不行就先砸Target端Unmap这个LUN，然后断开。

87. 针对某IO测试结果的分析

前几天测试了一台EVA4400，服务器系统Linux，使用Orion测试的，读写比例90:10。使用sar和iostat监控，其中发现了这种情况，请各位给解答下：EVA4400满配12块硬盘，我划了一个Disk Group，划分了4个vdisk，容量分别为几十GB，服务器与盘阵间有光纤交换机，Linux上打上了HP的多路径软件，系统中fdisk -l；系统盘是c0d0，盘阵映射到服务器中的是sda/sdb/sdc/sdd。测试结果如下。

第一次：

Orion测试值：0, 46041, 41442, 3712 3, 3637 4, 3588 5, 3479 6, 3411 7, 3310 8, 3206

sar值（部分）：

13:55:34	DEV	tps	rd_sec/s	wr_sec/s	avgrq-sz	avgqu-sz	a
13:55:34	dev8-0		959.74	13795.80	1564.84	16.00	14.72 1
13:55:34	dev8-16		1077.02	15539.66	1695.90	16.00	10.82 1
13:55:34	dev8-32		1184.42	17069.33	1868.53	15.99	16.09 1
13:55:34	dev8-48		1313.39	18961.84	2057.14	16.00	22.11 1

iostat值（部分）：

10/26/09 13:55:34									
Device:		rrqm/s	wrqm/s	r/s	w/s	rkB/s	wkB/s	avgrq-sz	avgqu-sz
cciss/c0d00		0.00	0.50	0.00	0.40	0.00	3.60	18.00	0.01 13.25 11.0
sda	0.00	0.00	862.50	97.70	6904.00	781.60	16.01	14.74	
sdb	0.00	0.00	972.20	106.10	7776.00	848.80	16.00	10.83	
sdc	0.00	0.00	1068.80	116.90	8544.00	935.20	15.99	16.11	
sdd	0.00	0.00	1186.00	128.80	9490.40	1031.20	16.00	22.13	

第二次：

Orion测试值： 0, 2470 1,874 2,650 3,653 4,634 5,623 6,597 7,597 8,580

sar值（部分）：

14:19:04	DEV	tps	rd_sec/s	wr_sec/s	avgrq-sz	avgqu-sz	a
14:19:04	dev104-0	1.20	0.00	26.43	22.00	0.03	21.50 7.25
14:19:04	dev8-0	136.44	1915.52	227.43	15.71	16.85	127.93 7.
14:19:04	dev8-16	148.75	2146.15	233.83	16.00	1.29	8.65 4.
14:19:04	dev8-32	165.17	2386.39	296.30	16.24	46.35	250.14 6.
14:19:04	dev8-48	186.29	2657.06	323.52	16.00	1.56	8.39 4.

iostat值（部分）：

10/26/09 14:19:04

Device:	rrqm/s	wrqm/s	r/s	w/s	rkB/s	wkB/s	avgrq-sz	avgqu-sz
cciss/c0d00	0.00	2.10	0.00	1.20	0.00	13.20	22.00	0.03
							21.50	7.2
sda	0.00	0.00	121.30	15.00	956.80	113.60	15.71	16.83
sdb	0.00	0.00	134.00	14.60	1072.00	116.80	16.00	1.298.
sdc	0.00	0.00	147.00	18.00	1192.00	148.00	16.24	46.30
sdd	0.00	0.00	165.90	20.20	1327.20	161.60	16.00	1.568.

问题：

（1）为什么**4**块盘第一次测试负载相同，而第二次负载不同？负载相同时**TPS**值较高，而负载不同时**TPS**很低？脚本相同。

（2）请问如何解决这种问题？在其他客户那里也见过这种情况，造成服务器整体性能很低。后又建立两个**vdisk**，添加到服务器中，发现同样会出现上述情况，**1**个负载**100%**，**1**个百分之六七十。

4个盘一起**IO**么？这样的话会造成所有磁盘全部忙于寻道，而且根据控制器内部的相关策略，**4**个虚拟盘之间平衡可能会被打破，所以有些高有些低。先单独测试每个盘看看结果，再一起测试。一条定律，如果在**1**组物理磁盘上生成了多个虚拟磁盘，而你又想同时对这多个虚拟磁盘进行并发的**IO**操作并且想达到较高的**IO**和带宽，那么重要条件就是提高**Stripe Depth**或者类似定义，越高越好。

补充一下，尽可能多的盘在一个**Disk Group**中，适合于上层单**IO**、连续大块**IO**的情况，非常适合于提高单**IO**的整体吞吐量。如果是多个虚拟磁盘并发上层**IO**，在一个**Disk Group**中加再多的盘，效果也提升不

了，但是有一个例外，Stripe Depth如果增大到可观的程度，盘越多当然越好。总之，Stripe Depth关系着上层并发IO的并发度，在并发IO环境中起着决定性作用。

88. 关于CDP厂商

现在做CDP的有多少家？

连续数据保护，记录每一个写IO并附以时间戳，在一段时间内可以恢复到任意指定的时间点，粒度不同，厂家不同，楼上说的是一分钟的力度，也有秒级，毫秒级，毫秒级一般也就够用了，因为你根本不知道哪个时间点的镜像是一致的。CDP相对于Snapshot的一个缺点就是很难保证一致性，只能通过牺牲RPO来保证一致性。目前飞康、EMC、IBM、Symantec都有做，其中飞康和EMC功能较强。

89. Snapshot与CDP在一致性保证方面的区别

请教冬瓜兄：（1）Snapshot是如何保证一致性的？（2）CDP为何不能保证一致性？原理是啥？

Snapshot在主机端有各种文件系统甚至应用程序级的Agent来强制将一致性的缓存数据写入硬盘之后立即做一次Snapshot。CDP是记录很细粒度的写IO，你总不能每秒强制应用或者文件系统将缓存刷到硬盘吧？CDP只能通过牺牲RPO来达到一致性，随便找个时间点恢复除非碰巧，一般是不一致的。所以需要一些回滚手段，如EMC RecoverPoint就是利用日志来回滚到上一个Commitpoint处。

是不是可以说是以镜像的方式进行数据保护的？

因为CDP底层实现起来比较复杂，在Online存储上直接实现对性能有影响，所以飞康等都是使用镜像的方式先将Online数据镜像之后再在镜像上实现各种功能。三种镜像方式：主机端Agent（比如LVM等卷镜像软件），Fabric API比如SANtap Service（FC交换机端口镜像），存储设备处对卷镜像。

对于**CDP**对数据保护的一致性方面有什么见解？

CDP的RPO在底层看似是0，但是对于上层一定是大于等于0，因为需要回滚，靠CDP自己的日志回滚也好，或者干脆靠应用层回滚也好。另外，关于Snapshot一致性是分层的，文件系统及其下层，还有应用层，前者做的Snapshot至少在文件系统层是一致的，但是不一定应用层一致，后者则可以保证应用层一致。还有，CDP得根据实现方式来判断是否是“真实的数据复制”。对于EMC RP或者Ipstor CDP这种镜像数据之后在镜像中做CDP，当然是复制了。如果其他实现方式比如直接在线实现或者使用附加的Block仓库方式，那也需要指针。

“所以基于磁盘**Block**设备的**CDP**，包括飞康的、**EMC**的，都是扯淡！”

底层只能是先保持在线存储的同步镜像，至于是否一致，这个又绕回来了，两个一致性层次：文件系统及其下层，应用自身的Buffer一致性和逻辑一致性，只有应用的逻辑一致性才是端到端的一致性，也是追求的最终目标。通过底层实现的CDP可以靠上层回滚来实现一致性，而如果从应用层来实现CDP，比如Oracle Dataguard，可以看做是一种CDP，它也一样需要应用replay。所以底层CDP和上层CDP，本质是一样的，不是扯淡。

90. 关于LAN、WAN、SAN、FC、iSCSI、NAS、LAN-Free、Frontend-Free

请解释一下：**LAN、WAN、SAN、FC、iSCSI、NAS、LAN-Free、Frontend-Free**

LAN: Local Area Network

这个都知道，局域网。但是好像大家都叫习惯了，反而有人不理解LAN的本质意思了。好好看看这三个字，“局域网”，你在你的本地站点看得见摸得着的所有网络，任何形式的网络，都属于LAN。比如：本地电话交换网络，本地以太网，本地存储网络。有人质疑，本地存储网络也属于LAN？存储网络不是SAN么？这就是一种概念不清并且不统一。将本地SAN归纳到LAN中将会很好地统一这些概念，随着本文的继续读者将会体会到。

WAN: WIDE Area Network，广域网

泛指跨越远距离的链路，或者地理上相隔很远的两个网络互联起来形成了广域网。广域网这个词在应用的时候大部分时候指代长距离传输链路，即WAN常用于指代长距离链路而不是网络。比如在说“将本地站点通过WAN连接到远程”，这里的WAN就是指长距离传输链路。为何WAN一词会被赋予这种指代，原因是因为长距离链路普遍都是点对点链路而不是多点互访型链路，所以一个广域网，其具有广域网意义的也就是局域网之间的这条链路了。你说以太网链路是否可以作为广域网链路，当然可以，FC链路也同样可以跨长距离，所以此时也属于WAN链路。

SAN: Storage Area Network，存储区域网络

即专用于传输对存储设备IO数据的网络。可以是任何形式的网络，比如FC网络、以太网络、SAS网络、Infiniband网络，或者，任何形式的IP网络。使用FC网进行IO传输的叫做FC-SAN，同理，使用IP网络的叫做IP-SAN。基于SAS网络的呢？当然叫SAS-SAN了，以此类推。那么，有人推出来个Ethernet-SAN，是不是也可以呢？上文说SAN也可以基于以太网络。没错，你可以这么说。你可以把FCOE称为一种Ethernet-SAN。那么IP-SAN是否也属于Ethernet-SAN呢？部分属于，因为IP-SAN的底层链路一般情况下都是使用以太网的，但是绝对不能说IP就是以太网。比如你使用ADSL通过Internet一样可以连接到iSCSI Target，那此时就不能称其为Ethernet-SAN了，只能叫IP-SAN。以上列举的这些访问方式，或者说协议，目前都是基于Block形式的SCSI协议+底层传输协议。还有另外的BlockO访问方式协议集将在下文描述。

NAS：CIFS和NFS以及其他第三方厂商自行开发的基于文件偏移量IO的文件型IO协议

业界将能够提供文件型IO访问而存储数据的存储设备称为NAS，即Network Attached Storage。NAS设备在局域网内部目前都是使用以太网来作为底层传输链路的，寻址路由和传输保障协议分别使用IP和TCP这两种目前最为广泛的协议。在这之上便是CIFS、NFS等上层协议了。NAS是一种设备，而不是一个网络。但是可以用NAS来指代文件型IO的访问方式，比如描述某个设备的IO方式是Block方式还是NAS方式。注意，避免用这种描述Block备使用SAN方式还是NAS方式，原因下文再述。

SAN所包含的元素

关于SAN这个词有几个误区或者不成文的说法，比如只把将基于

FC的数据传输方式叫做SAN，或者反过来，当说SAN的时候只表示FC-SAN而忽略了IP-SAN，这样容易对阅读方造成不便、混乱和误解。比如在描述某款产品的时候，“前端支持SAN方式访问”，其实这个设备只支持FC-SAN，但是他没有明确指出是否支持IP-SAN，这样就给阅读者带来了不便。

另外，SAN是一个网络，但又不仅仅只表示由交换机组成的底层网络，而它包含更加丰富的元素，比如磁盘阵列、磁带库、虚拟化设备、备份服务器等。可以说SAN这个词与SA（Storage Area）近乎同义了，或者说SAI（Storage Area Infrastructure）。比如可以这样说：“某SAN中包含两台xx磁盘阵列，一台备份服务器，一台磁带库。”

建议：今后在描述具体技术参数时杜绝单独使用SAN一词，代之以FC-SAN、IP-SAN、NAS-SAN等，或者干脆直接只用协议来描述，比如FC、ISCSI、CIFS、NFS。比如“前端支持FC、ISCSI访问方式”，这样就没有任何歧义了。

各种关系论

LAN与WAN的关系：LAN与WAN就是本地和远程的对应关系，本地的就是LAN，远距离的就是WAN。

以太网与LAN的关系：绝对不要把LAN等同于以太网。LAN是一个大涵盖，不仅仅包括本地以太网络。如果某人描述“客户端与服务器通过LAN连接，服务器与磁盘阵列通过SAN连接”，这句话不是不对，但是描述太含糊，“通过LAN连接”，哪种底层链路？没说。虽然大部分人还是理解其意思，但是终究不严谨。不如直接说“通过以太网连接”。另外，本地SAN也属于LAN，所以这句话追究起来，就是个病句了。

SAN与LAN和WAN的关系：如果一个SAN，或者说SA，完全在本地，则可以把这个SAN归属于LAN。如果某个SAN是跨长距离链路部署的，则可以说这个SAN跨越了WAN或者WAN链路。

SAN与FC和ISCSI的关系：FC泛指一种传输协议，即Fibre Channel。用FC协议来承载SCSI之后产生的协议叫做FCP。ISCSI指用IP网络来承载SCSI之后产生的协议。SAN与FCP和ISCSI的关系，就是网络与协议的关系。数据在SAN网络内传输时使用的多种协议中包含了FCP和ISCSI协议。

SAN与NAS的关系：NAS指一种设备，SAN或者说SA指的是一种网络 and 这个网络区域内的各种元素。NAS当然也属于存储区域中的元素，所以NAS属于SAN。

乱七八糟的Free

LAN-Free：这个词根本不应该存在。关于它的由来，是起源于备份领域的一种技术。一般在某个企业网络系统内，客户端与服务器是使用以太网进行数据通信的，而传统的备份系统中，备份服务器和介质服务器也需要使用以太网与需要备份的服务器进行通信，将服务器上需要备份的数据通过同一个以太网传输到介质服务器上从而写入介质保存。由于客户端与服务器通信的以太网络的繁忙程度直接关系到客户端的响应速度，本来对延迟就比较敏感，而备份数据流也通过同一个以太网进行传输，只能是火上浇油。那位说了，以太网交换机目前这么便宜，再买一个，服务器上再加个网卡，专门用于备份数据流的传输，井水不犯河水，客户端以太网不就是Free了么，不就行了么？是啊，所以说“LAN-Free”这个词真的让人摸不着头脑了。为什么这么说呢？上文说过，很多人都直接用LAN来指代客户端与服务器通信的前端以太网，或

者不管前端还是后端，就指以太网。这样的话，刚才那个例子，再用一个单独出来的以太网专用作备份数据流的话，这样到底叫不叫LAN-Free了呢？如果说，LAN-Free中的LAN仅指代前端以太网而不是备份专用以太网的话，那么上面的方法就是LAN-free；但是如果这个LAN指的是以太网的话，那么上面的方法依然用到了以太网，就不是LAN-Free。到底是不是呢？乱了。再说了，LAN本来的意思是局域网，即本地网络，不管你再怎么折腾，只要还在本地备份，那么都属于LAN之内，就不是所谓LAN-Free。彻底乱了。所以说，这个词根本不应该存在。哪个词更能表达这种不消耗前端客户端网络的备份方式呢？鄙人发明的一个词叫做“Frontend-Free”。

Frontend-Free: 前端网络Free，即不管你怎么折腾，只要不耗费前端网络资源，管你再用一台以太网交换机也好，或者直接使用FC网络来存取备份的数据也好，都是Frontend-Free。这不就解决了么？

笔者在做一些方案或者写一些文章的时候，都会时刻注意这些词的应用，力求清晰严谨。

91. RAID Stripe问题

我的疑惑，比如3个盘组成的**RAID**，条带**64KB**，如果我刚好有**64KB×3**的数据要写入，那正好一次搞完，是这样吗？

条带是横跨3个盘的，**64KB×3**的话，是一次写完，控制器对每个盘一个IO就能写完这些数据。

如果我的数据只有**64KB**呢，写还是不写？还是按**4**说的写到其中某块盘的**1**个条带中？

看来你没理解条带，你说了条带=64KB，你要写64 KB，那么当然是一次写入三个盘了。这里不考虑64除不尽的问题，最好弄成4块盘来举例。

如果再小点，只有**32KB**呢？写还是不写？怎么写？也是写到其中某块盘的**1**个条带中吗？那该条带剩下的空间意味着浪费了是吗？

“某块盘的一个条带”，这说法不对。条带由Segment组成，条带在每个盘上占用一个Segment。32KB就会占用两个盘（共4盘），控制器对每个盘一个IO。剩下怎么浪费了？

我一直也有这样的疑虑：如果是存储的话，会被先**Cache**，然后凑足条带的数据再写入？

Cache当然要Cache了，理想情况下控制器尽量整条写，也就是组合合适的IO。要是想研究这些底层东西的话，得从头梳理一下这些细节概念。

92. 关于IO冲突问题

小弟有一个疑问，一个主机系统对应一个**Array**还是一个**Array**能对应多个主机系统（只要控制好**Storage Partition**）？有这个疑问主要出于**IOPS**性能的考虑，若一个**DS 4700**有**16**块盘，若我建两个**Array**（除了**hotspare**），每个**Array**对应一个主机系统（不同**OS**），那么每个**Array**中的磁盘数就会减少，相应的**IOPS**就会降低；若我只建一个**Array**，将这个**Array**对应到两个主机系统，那么**IOPS**就能得到提升。但我不知道我的想法是否可行，请**DX**们指点。

最好是两个**RAID**组，每个组根据需求提供相应的磁盘数。同一个

RAID组提供多个LUN给多个主机，存在争用冲突，如果实在必须这样，建议将Stripe Depth调到最大以最大程度避免IO冲突。

93. 关于LUN分布方式的问题

我这里有个**DS4800**，只连接了一个扩展柜。共**16**个磁盘。我在这个**DS4800**划了两个**RAID 5**，每个**Array**里面建立一个**LUN**，名字分别为**LUN1**、**LUN2**，将这两个**LUN mapping**至同一台主机，磁盘分别为**hdisk1**、**hdisk2**。然后**Aix**里面在**hdisk1**、**hdisk2**上建立一个**LV**，这个**LV**通过**LVM**实现条带化。这样就实现了**DS4800**的纵横向条带化。我划分的**LV**是供给**OLTP**数据库以裸设备形式使用的。但是具体这两个层面条带化的大小应该怎么设置才合理呢？是不是这两个层面的两个条带化大小相同效果才最理想，那具体设置多大？请高手们指教。谢谢！

建议做成累加模式而不是**Stripe**。不过也要根据实际来看，如果两个**hdisk**的负载均会比较高，而且并发性比较高，那不推荐再次做**Stripe**。如果你非要说的话，那就将**Stripe Depth**调到最大以提供最大的并发度。

94. 测试的伪结果

怎么判断**Total I/Os per Second**和**Total MBs per Second**读数高了或低了，分别又是由什么原因引起的？

读数高低主要由两个因素决定，一个是目标存储的性能，性能高的当然高，低的当然低。但是还有一个最容易被忽略的因素，就是缓存，包括**IOmeter**所处的主机端缓存和存储端缓存，尤其是测试顺序读的时候更容易受到缓存的影响。比如第一次测试，读数可能较低，但是第二

次测试读数显著增高，就是由于缓存的影响。

选择**1个worker** 和选择多个 **worker** 去测一个**RAID**分区或**HDD**分区有什么不同？我用**4核CPU**选一个**worker**，**100%read**去测一个**5400**转的**HDD**时，**Total MBs per Second** 的读数会有**110**多，比较正常。但当我加到**3个**或**4个worker**时，读数就只有**50**左右了。

不知道你多个**worker**测试的是同一块物理硬盘的不同分区（地址段），还是多个硬盘。对于一个物理硬盘，或者一个物理**RAID**组，主机端的**IO**个数最好为**1**，如果同时多路主机端**IO**，会造成物力资源争用，导致磁盘不停寻道，从而显著降低**IOPS**和带宽吞吐量。当然如果多个**worker**对同一段地址进行读取，由于先读出的数据在缓存中，后来的**worker**读取同一段地址时，性能就会大大提升。如果就如我开始说的，多个**worker**对不同分区或者地址段进行测试，那么整体性能不升反降。有很多人反映过类似问题，我也都回答过，在一些论坛，可以搜索一下。

如果同时选**4个worker** 去测**4个**不同的分区，这种做法有问题么？

如上个问题一样，不能说是问题，只要是机械硬盘存储，都会这样。

传输数据块大小的选择对测试结果会有什么影响？另外为什么一般都选**4KB**、**16KB**、**64KB**、**256KB**、**512KB**、**1MB**等，而不是**3KB**、**5KB**、**20KB**、**100KB**、**200KB**等？

这里数据块大小就是每个**IO**的**Size**。从你的问题我推断你还没有深刻理解**IO**的概念和属性。在《大话存储》中我用了尽可能详尽甚至啰嗦的篇幅来讲述**IO**，如果你还没有阅读这本书，请尽快阅读。这里我还是

再重复的简要描述一下IO。IO的三大件：目标/地址段/读写的长度。关于IO Size和IOPS的关系，请同样阅读《大话存储》。不同IO Size和IO属性（顺序、并发、连续、随机）对结果有不同影响，同样，阅读《大话存储》。为何IO Size都是2的幂次，因为计算机领域普遍习惯用2的幂次，而且每磁盘扇区一般都是512B，为了扇区对齐。每个IO中的“读写长度”都是扇区的2的幂次倍。

95. RAID级别问题

最近看了一下大话存储**RAID**级别，有几个问题不明白：（1）就**RAID 5**，条带大小为**128KB**，如果写入的文件很小如 **2KB**，那存放的方式和占有容量是怎么确定的？剩余条带怎么处理？（2）磁盘在格式化的时候有个单元分配大小，这个好像就是文件存放的最小单位，和**RAID**里面的块有什么区别？

要彻底理解这个问题，我建议楼主在文字的基础上勤画图。

首先你要写入2KB的“文件”，既然牵扯到文件，就要牵扯到文件系统的分配单元，也就是你第二个问题。**RAID**向操作系统提供的是卷，是连续的扇区（或者叫LBA）空间地址，而卷向文件系统所展示的是分区，相对于卷，分区就是将大片连续地址再次切开。

文件系统用扇区组成所谓“簇”，或者叫cluster，或者叫分配单元，FS存放数据只以簇为单位，而不会出现“读入或者写出半个簇”这种IO命令。而卷之下的各层都可以以单个扇区为单位了，扇区是存储系统最小的IO单位。

明白了底层这些映射关系，就可以着手研究IO行为了。再说回来，

你说要写入2KB的“文件”，当然是要调用文件系统进行写入了，如果文件系统的簇大小被设定为2KB，则文件系统会在分区内分配2KB的空间来存放文件实体数据，也就是4个扇区的连续地址。此外，还需要更新inode（unix）或者MFT表（Win）等所谓“元数据/MetaData”，也需要占用额外的空间。

值得一提的是，inode或者MFT对应某个文件的metaData本身是可以存放一定数量的文件实体数据的，一般是64B，小于这个数值，文件直接存放在inode中；大于这个数据，inode中会被加入二级和三级扩充映射指针指向文件实体数据被存放的扇区地址用于寻址操作。再来看看RAID所提供的角色，刚说了RAID提供一片连续扇区地址，假如文件系统选择了扇区1024~1027号这4个连续的扇区作为这个2KB文件的存放空间，则文件系统写入这个文件的时候，卷接收到这个指令之后，会将这段扇区号码传送给RAID驱动，RAID驱动接受之后会将号码传送给RAID芯片进行地址翻译（硬RAID卡）或者直接在驱动层面进行地址翻译（软RAID卡）。地址翻译这里的意思是将这段连续的地址映射到实际的物理硬盘地址，因为RAID提供给上层的是虚拟磁盘/LUN，一个LUN可以分布在多个物理硬盘上。

地址翻译的过程一定要查询Stripe也就是条带映射表，当初你怎么分的，此时就会影响翻译之后实际的硬盘扇区地址。再回来说你的RAID 5的128KB条带，128KB条带=磁盘数量乘以每个磁盘上组成这个条带的Segment大小，也就是说一个条带把排列的多个磁盘横向切成了一条一条的，硬盘本身相当于竖条，而横条和竖条切开之后形成的小格子就是Segment，也叫条带深度，Stripe Depth。

比如8个盘的RAID 5系统，其中一块用于存放parity，128KB条带除以8等于16KB，也就是说Segment=条带深度=16KB=每个磁盘上贡献

一个条带所使用的空间。再回来说2KB的文件写入，这个情况下，地址翻译会将2KB的地址翻译为“磁盘m上的n到磁盘1上的n+3号扇区”，当然也可能是“磁盘x上的y到磁盘a上的b”，总之地址落在物理硬盘上的哪个区域，如条带中央/条带边缘，单个物理硬盘/多个物理硬盘，文件系统是不知道的，由RAID层面决定。这种不知道称为“盲”，现在大多文件系统都盲，但是也有不盲或者半盲的。

我们在RAID层面来设计条带，分步等，实际上都是盲操作，效果不会很大，包括RAID 5本身的并发IO特点，也是盲并发。所谓并发，拿你刚才的例子，某个IO需要写入2KB的数据，如果地址翻译结果为这4个扇区落在一个Segment里，则这个IO只会占用一个数据盘，外加需要占用parity区域。如果此时还有一个IO需要写入2KB数据，而这次的4个扇区落在了另一块数据盘上，而它需要的parity数据恰好与前一个IO及其所需要的parity区域不在同一个盘上，则这两个IO可以并行操作，4块磁盘同时读写。由于这种并发是基于“恰好”的，所以RAID 5提供的是盲并发，要实现不盲的并发只能靠上层文件系统。说了这么多，等于给《大话存储》做了一个注释吧。楼主要是想彻底把这些基础知识扎实打牢靠，请阅读《大话存储》。

96. AIX中磁盘的Queue Depth问题

对于AIX系统，磁盘的属性里有一项是**Depth**，默认值是**10**，对于裸设备应用，调大**Depth** 性能会有所提高，请问**Depth**是什么意思？是不是调的越大越好啊。

IO队列深度，即Queue Depth。与hdisk底层对应的存储所提供的LUN所对应的队列深度对应起来即可。

97. RAID 10磁盘数量与性能的关系

想请教一下大家**RAID 10**磁盘数量跟性能的关系。如果从原理上分析，是磁盘越多性能越好，但实际是这样么？**RAID 10**最多能做多少块盘呢？经验值一般都是多少呢？

只要控制器允许，盘越多性能越好，但是多LUN并发环境下，整体性能提升有限，不如单LUN环境。据我所知IBM DS4新固件以及DS5可以达到30盘的RAID 10。

98. VMware ESX是如何使用SAN存储资源的？

本人对**VMware ESX**不是很熟，请问它是如何使用**SAN**存储资源的？

（1）是在磁盘阵列上分配一个大尺寸的**LUN**给**esx**服务器，然后通过虚拟磁盘的方式再分给各个虚拟机（**vm**）吗？

（2）还是在磁盘阵列上配置一个个的**LUN**，再直接分配给每一个虚拟机（**vm**）呢？

你说的两种方式都支持，前者是VMFS方式，后者Raw Disk Map方式。前者在一个**LUN**上创建多个**vmdk**文件给多个虚拟机使用，后者一个**LUN**只能给一个虚拟机。

99. 如何快速备份大量小文件？

某个项目中遇到的问题：几十个**TB**的数据，都是电子文件，大部分大小在几百**KB**到几**MB**之间，一经写入，很少改动。目前有磁带

机，一盘磁带大概能写**800GB~1.5 TB**。这样的数据，怎么设计备份机制比较合适？请大家指点。

可以先直接采用持续复制方式镜像到外部存储D2D即可，Snapshot等在镜像中来做。大量小文件备份恢复，这种d2d持续复制方式是最佳的。或者直接备份整个LUN空间也是可以的。

100. 如何理解IO惩罚？

之前看《大话存储》，有些疑问。

（1）我理解的写惩罚，是该模式下某些情况写数据因为校验盘等因素导致**IO**次数相比正常增加，不知道这么理解对不对？像**RAID 5**的写惩罚高，主要是因为针对小数据的写操作？

（2）据说**RAID 5**的读改写模式效率低，是因为它要多出两次**IO**，一次是读出校验盘数据，一次是写回校验盘数据，加上它读出和写回需要改写的的数据，一共是**4**次。为什么校验盘不能和数据盘一同读出和写入呢？它们本身在两个磁盘，应该可以并发的？

这里当然是可以并发**IO**的，但是为什么你非要将这两个并发的**IO**理解为“一次**IO**”呢？

我的意思是既然可以同时读出或写入数据盘和校验盘，那么读取写的**overhead**体现在那里？

如果将这个“同时”用于处理其他上层**IO**请求，而不是用来读写Parity或者其他segment上的数据呢？

101. 如何分布LUN的问题

讨论下，**HDS** 的存储，现在给数据库划**8GB**大小的**lv**，是单独在一个**pv**上划**lv**，性能好，还是在多个**pv**上划**lv**的性能好？（前提是不做条带化）

这个问题的大方向是：不要管**lvm**，这些都是虚的，也不要管存储层什么这个**ldev**，那个**vol**，这个**LUN**的。我们最终就看它用了多少块物理磁盘，也就是你能看得见的磁盘。划了几个**RAID**组，你的应用是否是多进程并发访问同一组磁盘，而且这多个进程又不相关，比如就是两个完全独立的系统，那么知道了这一点，你再对应自己的需求，一组磁盘提供的性能是恒定的，多人来抢就不如一个人用的爽，就这样。任何问题，不管哪个层面的，用这个方向来判断，屡试不爽。

102. 磁盘内部传输率、外部传输率、传输带宽、接口速度之间的关系

在本书第3章说道磁盘的内部传输速率、外部传输速率、传输带宽、接口速度之间有什么关系？另外**Ultra 320 SCSI**传输速率可以达到**320MB/s**，这个**320MB/s**指的是上述的哪个速度？

内部是最大理论速率，外部是最大实际理论速率，也就是被寻道打断之后的速率，接口速度是出口速度，可能会卡住外部速率。理论传输带宽=接口速度。实际传输带宽，瓶颈在哪就等于哪。**U320**指接口速度。

103. RAID 5对小文件的性能提升问题

多块SSD磁盘做RAID 0/5，在小文件方面，对性能提升的作用是正面的还是负面的？以我的理解，RAID 0/5对小文件的性能提升是负面作用。

多块SSD做RAID，与机械硬盘RAID无二致。提高并发，不管什么介质，都有益。不知道为何理解RAID 0/5小文件的负面作用。

对于大量小文件，比如10KB左右的。RAID条带大小一般4、16、32、64、128 和 256KB，多数情况下64KB，小文件基本上不分片，RAID 0没有优势，RAID 5还得计算校验值。因此，我觉得还不如单块SSD性能好。不知道这样理解对不对？

谁说一定要分片才有优势的？8个盘RAID 0，每个盘服务一个文件IO，并发8个，这样很好。有时候分片反而性能差得一塌糊涂。另外，用SSD做RAID别老想着分片，没一点好处，除非是大块连续IO。对10KB也分成1KB的片？不怕影响SSD寿命么？分片以后，所有SSD寿命都被摧残了，还不如就着一块弄。SSD组RAID最怕就是因为Parity分布太平均而造成多块SSD在短时间内全坏，所以针对SSD的RAID需要一定的Parity分布算法，不能太平均，得一个一个坏，一个一个的换上，换上新盘之后还要重新分布Parity。因为如果不重新分布Parity，新盘的损耗可能会追赶上其他盘的损耗而达到同一段时间内损坏。这种SSD的RAID算法目前国内高校有些课题在弄的。

RAID内部，数据分片应该是自动的，超过条带大小就分吧。还是说，分不分片是可以通过设置来控制的？

10KB，可以明确告诉你，除了RAID 3之外任何RAID都不会分。分不分可以设置Stripe Depth，不过Stripe Depth自身也有限，分不分得看具

体情况了。

104. 集群NAS与并行文件系统的区别

集群**NAS**与并行文件系统的区别是什么？都是在集群方式下实现的，有什么不同呢？

现在集群NAS和并行文件系统有融合的趋势。包括**Ibrix**、**Stornext**、**Panasas**等，它们都是在并行文件系统之上提供NAS抽象层的，所以你说它是并行FS也可以，说它是集群NAS，也可以。

105. 快照、容灾如何保证数据一致性的问题

存储的快照、复制技术如何保障数据库的一致性？据说实测时，**Oracle**不一定能**Open**。但又有很多容灾的案例，不知用的什么机制保障数据库的一致性？

快照如果需要保证一致性，那么第一层可以在FS层进行Flush之后做快照，这样可以保证FS一致性，但是不能保证应用程序层的一致性，要做到后者，那么需要在应用层进行cleanup，然后做快照。实现着两层的一致性需要在主机端安装一个Agent来与FS或者应用通信协调，在用户决定执行快照时，通知FS或者APP来执行Cleanup。**Microsoft**提供了VSS服务，所有不同厂商的应用、存储、Agent都可以基于这个服务平台来实现APP层面的一致性。至于容灾，容灾时只要保证完全同步，那么就不会有问题，但是代价也是巨大的。在异步时，只能够通过一致性组的方式来保证时序一致性，但是却不能保证上层逻辑一致性，如果连时序一致性都无法保证，那么Open成功的几率会很低。

保证了时序一致性的前提下，能不能Open，也要看造化，能Open，则通过Replay来恢复一致性但是却丢失了数据，如果还不能Open，那么只能是恢复到容灾端最后一个绝对一致的快照了。

106. 网络IO问题

我今天再次测试了我的服务器，发现了一个现象：我**1.5TB**硬盘里有**10个20GB**的图纸，首先我把这个硬盘连接在服务器上，通过我本地的电脑将这**10个**图纸中的两个同时分别用光驱进行刻录。当只开启一个的时候，速度正常，当另一个光驱开始刻录的时候，两个光驱的速度立刻下降甚至到了**0速**的状态。于是我把硬盘从服务器上拿了下来，直接连接到我自己的电脑上，再次用同样的方法对文件进行刻录，这个时候效果立刻不一样了：同时刻录的两个光驱都可以按正常速度进行工作，没有出现速度下降问题。特别说明一下，我与服务器之间的连接是通过千兆网卡连接，用**FTP**测试过速度，基本上都在**80MB/s**左右，所以应该网络不是瓶颈。请大家帮忙分析下问题所在。个人想法，我想会不会和服务器上的主版或者服务器的设置有关呢？我用的是**Win2003**数据中心版，没有进行多余的设置，基本上就是安装好后给目录做下共享，是不是有什么需要设置或者遗漏的呢？

刻录对数据流波动是很敏感的，1G的速率不能和本地总线以及硬盘接口速率相比。加之cifs的开销很大，当另一个进程也连接cifs目录打开文件时会产生大量数据包，严重堵塞网络，刻录程序感知到数据的这个波动，当然就降速了。用SMB 2.0协议会有所改善。Windows Server 2008和Vista/ 7已经使用SMB 2.0了。

107. Lustre与pNFS之间的异同

哪位高人能讲解一下**lustre**和**pnfs**之间的异同？

lustre和pnfs都使用OSD的概念。传统nfsv3和cifs做不到并行集群访问的要求。所以nfs 4.1也就是pnfs意识到了这一点，将架构改成与lustre等类似以获得并行访问。这就是osd与传统nas最大的不同之处，类似的还有PanFS、Ibrix的FusionClient等。

Lustre是一个并行文件系统的名字，而pNFS则是用于并行访问的一种协议。一个是文件系统，一个是协议。

108. 阵列之间的远程容灾复制，其底层到底是怎样的？

通用做法以及点出几个我所知道的厂商的做法，供参考。

（1）远程复制可以通过多种链路进行，如果通过fc进行，则一定都是使用ILT模式的，也就是initiator-target-LUN。但是有一种除外，下面会讲。

（2）本地阵列是否会直接挂远程阵列的LUN上来比如当做/dev/sda这类东西来做，不一定，一般都不会，都是直接与IO设备驱动层交互直接对LUN进行操作以绕过块设备层以及其他各层，提高效率。

（3）推和拉模式。一般选择拉模式，及远程主动向本地要数据，表现为读过程。为何这样考虑，是因为scsi write过程交互的数据帧比读要多，浪费链路资源。

（4）传输方式：标准scsi read/write cdb模式。

（5）控制模式：将控制数据封装到scsi cdb中传输，对端提取对应

数据进行定界、分析、处理从而控制数据传输。

（6）如果通过IP进行，对于一些大阵列来讲一般做法是将fc再封装到IP，比如弄成fcip，加fcip交换机之类。另外一种则是使用私有协议接口来完成镜像操作。

（7）使用私有协议完成镜像操作，这个就多了，很多基于IP复制的。

（8）厂商举例：emc、hds、ibm这类使用fc链路进行镜像的，都是使用ITL模式。netapp使用IP复制，使用私有接口，先将待传输的数据块进行描述、封装，传输到对端，对端提取分析之后发起拉取数据的操作。其协议并不是scsi/iscsi，私有协议，netapp也可以使用fc链路复制，但是由于采用私有协议，fc并不提供私有协议的接口支持，所以引入了FC virtual interface，及FCVI卡。这种卡可以提供FC上层的VI接口，从而方便地将私有协议转换成VI协议。其他也有厂商使用iscsi进行复制的，与fc复制的区别就是链路不同而已，就不列举了。

（9）至于阵列可以做initiator模式，当然可以，其中netapp的产品可以对所有口的模式进行翻转，翻转之后重启即可。这一点做的很方便，希望国内厂商借鉴和学习其技术。

109. 请问两台存储设备可以像交换机路由器一样做热备么？

我记得某路由器上面的主控模块，是叫Supervisor Engine吧，是可以插两个来做热备的。存储设备一般都配有双控制器，它们之间是可以热备的。但是对于两台独立的存储设备，即便是同一厂商型号的设备（每台设备都有两个控制器互为热备），它们之间也不能做热备，除非

是集群存储系统，多台独立节点之间是可以热备的。如果你非要让它热备起来，唯一办法是通过主机端的某种镜像软件，比如LVM，软RAID 1，或者一些厂商提供的专门在主机端对两个卷进行数据复制（远程或者本地）的软件，这样一台阵列宕掉之后，相当于主机端找不着其中一个卷了，主机会切换到另一个镜像卷继续工作，做得好的话透明度很高，做的不好的话，也得等比较长时间切换。比如有些人测试过LVM的镜像，结果发现一分多钟没切换过来的情况。

110. 请问iSCSI与FCoE究竟谁会成为主流？

这个问题很好。目前的现状是，iSCSI硬卡只有Emulex和Broadcom（尚未发布只是公开）的CNA卡支持，后续可能会有更多的厂商推出万兆iSCSI硬卡。如果用软iSCSI initiator的话，一个10G口子会耗掉30%左右甚至更高的CPU，就算用最新的Intel CPU也是这样，所以基本上4个10G口就会耗死一台阵列。但是FC卡就不会有这种情况，FCoE同样也不会。退一步讲，如果10G的FCoE与硬10G iSCSI比的话，卡件与协议本身来讲都差不多，成本也差不多，所以就要看外围辅助设备的成本，一台FCoE交换机目前来讲还是远贵于10G以太网交换机，所以要是小规模部署，甚至不如用8G FC交换机划算。其次是看看场景，FCoE更适合想融合之前已经部署的FC与新部署的以太网的场景，也就是大型久建的数据中心，对于新建数据中心，这个趋势还不明朗，笔者之前和Emulex的一名员工交流过，他的意思是新建数据中心他们推荐使用iSCSI硬卡，但是我估计这种说法是含有很大水分的，毕竟Emulex是目前仅有的一家提供硬iSCSI卡成品的公司，他们推荐iSCSI可能有一定市场目的。但是我个人看法，FCoE目前Qlogic、Brocade和Emulex都有产品了，为何10G的iSCSI硬卡只有Emulex一家产品，证明FCoE今后可能会有一波行情。FCoE和iSCSI，谁O不是O？区别就是一个是FC，一

个是TCPIP，抛开以太网，单看FC和TCPIP，前者高效但是扩展性差，后者效率稍低但是扩展性很好，其实这已经与以太网无关了，还是最后到底是认同FC还是认同TCPIP的问题。我的看法是，FCoE会弄出一波行情，但是FC Fabric这个协议很邪门，它要求交换机也要参与Fabric的建立，而且交换机起到至关重要的作用，这增加了复杂度并且降低了兼容性；而iSCSI却不要求交换机有什么上层协议智能。最后iSCSI有可能会替代FCoE。FCoE与iSCSI，厂商也尚未看清，谁也不敢冒然选路。

111. 所谓计算与存储合体，其产品形态是什么？

冬瓜头，快来...存储和计算结合之后，是什么样的产品形态啊？计算，存储，都很牛叉的机器？**JIM GRAY**，有一篇论文谈到，真正成本最高的地方在网络...我们也确实可以感觉到，在同步数据，或者做灾备的时候，最头疼的还是两个节点之间的通道有多大。

这种统一之后到底是个什么机器？答案是不是单独的机器，就是一群机器，通过软件模块联系起来，对于计算机来讲，硬件属于物质本源，属于阴，属于形；软件则属于精神本源，属于阳，属于神。用软件模块将计算和存储颗粒汇总起来发挥作用，并且将原本的以计算为中心的计算方法变为以存储为中心的计算方法，把计算颗粒分配到存储了计算所需要的数据的节点上，在哪存储就在哪计算，大幅提高效率和速度，避免了频繁大量数据传输，这也回答了你的另外一个问题“成本最高的是在网络上”，其实这句话暗指，数据移动起来成本太高了。网络本身成本不高。但是如果容灾，依然可以使用这个思想，即在哪存储就在哪计算，可以在业务层面进行双份，而不是数据层面，比如一笔交易，可以在业务层面将其同步到远端，远端针对这笔交易生成自己的数据然后下盘。一个实际例子是，比如数据库日志

同步方式的容灾，同步量相比直接底层数据同步来的少很多。

你说的那种“计算与存储都很牛的机器”，也不是没有，但是还不到时候，到了量子计算和分子存储时代，那时候计算机形态又会轮回到初始原点状态，单台机器，确实很牛，大家都拿高速网络来连接到这台超级计算机上获取资源。

112. 大量小文件的存储场景，有什么优化办法？

其实很简单，可以参考Google的GFS以及变种HDFS、淘宝TFS以及腾讯Tencent FS的设计。这些都是处理大量小文件的典范。大家知道传统的文件系统下，每个文件都要被创建对应的inode之类元数据，但是在海量文件场景下，传统FS已经无法承载如此多的元数据IO量以及如此庞大的元数据搜索计算量了，唯一的做法就是降低元数据量，那么势必就要降低文件实体的数量，所以这些文件系统无一例外的都是用了这样一种变通的方法，即在文件中再创建文件，比如一个64MB的大文件，比如其中可以包含16384个4KB的小文件，但是这个64MB的大文件只占用了1个inode，而如果存放4KB的文件的话，就需要16384个inode了。那么如何寻址这个大文件中的小文件呢？方法就是利用一个旁路数据库来记录每个小文件在这个大文件中的起始位置和长度等信息，也就是说将传统文件系统的大部分元数据剥离了开来，拿到了单独的数据库中存放，这样通过查询外部数据库先找到小文件具体对应在哪个大文件中的从哪开始的多长，然后直接发起对这个大文件的对应地址段的读写操作即可。另外还可以创建索引以加速文件查找动作。

在一个海量分布式文件系统中，元数据就像上面的思想一样是分级的，中控节点，也就是MDS，存储一级元数据，也就是大文件与底层块的对应关系，而数据节点则存放二级元数据，也就是最终的用户文件在

这些一级大块中的存储位置对应关系，经过两级寻址从而读写数据。其实这些一级大文件，就可以认为它们是卷了，也就是在卷管理层之上再存放文件，这样就降低了单一空间下的文件总数量从而提高性能。

113. 能否列举一下存储系统中所使用的芯片？

好的，知无不言，言无不尽！我就从头讲起吧。首先硬盘上的芯片就不说了，大家都知道，Micro Controller、dsp等。一个硬盘扩展柜中的所有磁盘都被连接到一个芯片中，这个芯片位于扩展柜的控制板中，早期产品用PBC芯片，最经典的就是Lattice的7147。PBC也就是Port Bypass Circuit，完全的FCAL环逻辑，后来发展为SBOD，Switched Box of Disks，其实就是芯片变了，也就是底层变为点对点直达物理布线，而上层协议逻辑依然还是FCAL，只不过仲裁的时候再也不用手拉手传递了，直接由控制器独裁，指哪打哪！这类芯片包括PMC8738、LSI等，PMC公司称这种SBOD芯片为Cut Through Switch（CTS），有些方案则混用PBC和CTS，CTS接磁盘，扩展柜之间则使用PBC串联，以上是FC方案。SAS方案就痛快了，全是SAS Expander，SAS交换芯片，没啥好说的，芯片之间采用4PHY宽端口一般。另外扩展柜控制板上还有其他ASIC，比如用来控制整个扩展柜的中控芯片，包括SES（Scsi Enclosure Service）逻辑处理，探测各路Sensor并用SES上报，再就是ROM和RAM芯片了，一般中控ASIC里就集成了ROM。RAM则一般外置，比如128MB单片。控制板上还有PHY，这个就不说了。再往上就是HBA上的芯片了，PHY，不说了。再就是IOP（IO Processor）或者SOC（System On Chip）等，负责处理核心协议逻辑，比如FCP、SAS等，带RAID功能的卡，这些IOC或者SOC就更复杂了。再往上，就是控制器主板上的特殊芯片了，CPU等不说了。往往一些厂商为了体现差异化，搞一块ASIC那是倍有面子！比如3PAR的号称Thin Built-in的Gen3

ASIC，HDS在AMS2k上使用的ASIC，IBM的DS5k（O的LSI的）上所用的ASIC，HDS高端存储中的ASIC。无非就是加速数据处理。

114. 可否把数据库中数据在磁盘的分布讲一下？

数据库方面我不行，只能讲一讲肤浅理解。

（1）传统关系型数据库。两种方式：数据库自己管理裸磁盘从而直接存放数据文件（比如Oracle的ASM），Oracle甚至有自己的OCFS集群文件系统（为何不叫它分布式而叫集群？因为它底层是共享存储的）用来支撑其RAC集群。再有一种就是数据库利用操作系统提供的文件系统（关掉缓存，也就是调用时采用Direct IO模式，AIX下还提供一种CIO，也就是Concurrent IO模式从而不让文件系统底层对文件加锁，DB自己有锁）来存储数据文件。

（2）Nosql系统。比如Cassandra的实体数据，就是sstable、commit log和bloom filter以及index file。这些文件底层一般就是Linux下的EXT3文件系统。这些文件都是连续地址写入的，避免随机写入，所以提高了性能。谷歌的Bigtable底层使用GFS分布式文件系统。淘宝的TFS在1.0版本时使用EXT3结果发现性能不行所以1.3版本时使用自己研发的本地文件系统，猜测一定是个轻量级的，因为TFS层已经考虑很多格式了。Windows Azure后台数据库则使用普通的SQL Server组成集群切片，各个节点之间使用日志进行同步，保证了分布式事务一致性，其底层就是NTFS文件系统了。

说点题外话，Oracle RAC这种方式类似于CPU的SMP；Nosql分布式集群类似于MPP。对于商用存储，像各种集群文件系统比如Veritas Cluster Filesystem、Stornext、中科蓝鲸BWFS以及基于类似架构的集群

FS所构建的集群NAS，就类似于SMP；像EMC的V-Max存储，则类似cc-NUMA（数据分布但是内存共享）；IBM的XIV、Infortrend ESVA、Dell P6000等多数Scale-Out的存储，则类似MPP。从趋势来看，SMP->NUMA->MPP，存储也是一样的趋势。

至于数据文件中的亚一级的数据分布，我仅了解Oracle是用extent来分配。几年前弄过DB2，可惜都忘了。所以这方面还得DBA来解答。

115. 为何灾备端的IO时序必须保证？

数据容灾的灾备端为什么要在多流TCP传输时保证IO时序，你指的是同一个应用的IO写入通过TCP后变成乱序了？问题是这是文件系统和块设备驱动保证的问题呀。

如果把一批数据用一个TCP连接发送到对方，是不会乱序的，TCP不会对数据流进行定界，但是绝对不会乱序，本来是1234，传到对面依然是1234，对方只需要从头将数据流解析定界出一个个IO然后刷入灾备端的存储介质就可以了。但是多流TCP的话，一批数据，并行切分由多个TCP连接发过去，此时多流之间的数据就乱掉了。

文件系统和块设备驱动不会保证IO的时序性，典型的比如Linux下的io scheduler，他就是可以乱序IO，为了优化，同样，底层磁盘的NCQ、TCQ，也是乱序IO的。解决乱序IO是上层应用需要考虑的，比如绝对不会在发起写请求而没收到OS内核返回信号之前再对同一个地址发起读请求，因为会造成过期读。但是这种由应用保障的方法是强实时性有状态的，而对于灾备端来讲，它的应用程序就是主站点的存储系统，而主站点存储系统是无法感知应用逻辑的，所以灾备端只能乖乖的按照顺序执行，否则恢复的时候会造成严重不一致，可能导致应用无法

启动，甚至连卷、文件系统都无法被Mount。

116. 一个文件系统问题

NTSF文件系统对数据的管理是基于一个叫**MFT**的主文件表，我想问的是，这个表在磁盘的什么位置是在**C**盘么？如果我分了**4**个区，那么要有**4**个**MFS**么还是一个**MFS**对应**4**个卷？我可不可以把文件系统认为是一个应用程序，如果有**FAT32**，**NTFS**等多个文件系统，那么开机后是不是这几个应用程序一直在运行？对于**Windows**和**Linux**等操作系统没有虚拟文件系统吧，之前我发过一个帖子说过虚拟文件系统，但是好像说错了，只有分布式文件系统才有虚拟文件系统之说，没有分布式文件系统的应用就没有虚拟文件系统之说。

每个盘的固定位置，有一个根入口，而且会备份多份，这也是一些数据恢复工具扫描时读取的，如果备份的**mft**都损坏，那么扫描工具会扫描残缺的元数据**tree**或者叫链也行，来重组数据。每个分区是一个独立的文件系统。可以认为是一个程序，记得以前就回答过，**ntfs**和**fat32**可以同时存在，处于**os**的内核区，两套代码共同运行，你挂什么他就起什么代码。**Linux**下的目录表现方式，一切皆目录和文件，就是**vfs**，这也就所谓的虚拟文件系统，可以将虚拟文件系统理解为目录，而实际文件系统理解为文件-块映射层，**vfs**是再往上封装的一层。**win**的表现方式和**Linux**有些不同，**win**是以分区为入口，而**Linux**是以/为入口，下面挂的可以是任何地方的任何文件系统，但是**win**下也可以将某个实际文件系统挂到某个目录下，这也属于**vfs**。**vfs**与是否分布式没有任何关系。

那么所有的**MFT**都要放在**C**盘里面么？另外对于某个分区格式化后比如**D**盘，建立的位图、索引等各种表都在**C**盘里或者**MFT**里，具体

D盘里面其实什么都没有是吗？

每个分区都有一套完整的元数据，C和D是独立的。分区表用来区分C和D的地址空间，这两个空间上的FS是完全独立的元数据，但是OS内的文件系统代码是一套，挂起多个分区上的文件系统。

117. 关于LUN与文件

NetApp除了做**NAS**以外，还有**SAN**存储，但是好像其实现方式是在**NAS**的基础上模拟出来的**SAN**。不知道谁清楚这个，能否给简单讲讲。另外，有些存储产品在底层使用了五花八门的技术，比如一些**LUN**的技术，我总有个疑问，这些**LUN**技术好像是在某种文件系统之上做的，否则不可能有那么大的伸缩性和灵活性，明白人请讲讲！

NetApp就是把文件当成**LUN**。而且他的**Volume**也是一个文件。所以了，不知道**VNXe**底层是怎么搞的，据**NetApp**说底层也是个文件，但是**EMC**不承认，还专门出了官方FAQ。

对文件系统又有点新认识，文件系统不仅是组织数据的规则，而且是一种权利。只有文件系统才能有组织的调动数据，现在想想有的**LUN**功能可以智能的让数据分层这里确实有值得怀疑的地方，也就是说底层数据越权造反了！

其实迄今为止最灵活的底层数据空间管理技术就是文件系统。其实文件和块本身的意义是相同的，两者都是将大空间再次分割为上层容易理解的，灵活的小空间。所以说，当**LUN**从树上下下来开始直立行走的时候，才发现其实文件系统早就变成人了。

118. 磁盘阵列中包含哪些协议控制器

请问磁盘阵列控制器里面是否内置了**SCSI ATA**控制器？我想磁盘阵列控制器里面内置**RAID**控制器、**FC**适配器、**iSCSI**适配器、**SCSI**控制器、**ATA**控制器。控制器和**Enclosuer**里面的磁盘之间是通过**FC-AL**架构连接的，**FC**只是网络传输协议，**FC**包的有效载荷部分都是**SCSI**指令和数据请问是这样吗？

其他都没问题。但是**ATA**控制器的说法不对，一般没有**ATA**控制器，也没有**SATA**控制器。**SATA**盘一般是通过**SAS**转接板或者**FC-SATA**转接板连到扩展柜的。**SATA**转**SAS**的话，**SAS**控制器会原生支持**SATA**协议。而用**FC-SATA**转接板的方案的话，控制器不变，扩展柜中使用**FC**转**SATA**的协议转换芯片。

附录2 IP硬盘——玩玩还是来真的？

很久之前，我记得大概是在2005年，听说过国外一家初创公司做了一种以太网口的硬盘，当然硬盘本身不是他们做的，他们只是在硬盘上加上一个转接板，专业说法叫做Dongle，实现基于以太网的SCSI协议传输，至于协议是否使用的就是iSCSI无从而知，对其具体细节也很不了解。

当时存储技术在国内还没有怎么得到重视，所以感觉这种东西非常新鲜，竟然可以这么玩！当然，这东西终究没成气候，逐渐淡出了业界。那时候，ATAoE、SCSIoE这种类似协议也一直有人在做，其目的就是抛弃TCP/IP这种厚重的传输协议而转为一种轻量级的适配到以太网的协议。但是目前来看，这类协议最终也没得到推广。

固态存储——崛起！

9年过去了，这9年里发生了很多事情。首先，企业级存储系统在国内得到了铺天盖地的应用，从一开始曲高和寡到现在的遍地开花，各厂商的企业级存储系统产品在这期间至少经历了4次升级换代，高端产品则经历了两次。

其次，在业务层面，主机虚拟化的崛起、基于虚拟化之上的云计算架构的崛起，互联网后端架构的变迁，海量存储和大数据分析挖掘系统的广泛应用，这三大变革性事件对很多存储技术、产品、厂商及生态产生了重大影响。

再次，底层技术不断革新，固态存储技术崛起，2012年应该算是SSD元年。业务和底层技术的变革，驱动着企业级存储做出一轮又一轮的变化，一开始是内部架构的变化，比如Scale-Up到Scale-Out，然后就是访问协议的变化，除了块和文件，对象接口越来越被广泛使用，再就是数据管理上的变化，企业级存储其实对固态存储介质是爱恨交织，明知道这小东西一定会颠覆自己苦心建立的基于机械盘的生态系统，但又不能不迎合潮流，出现了各种数据分层分级方案和技术，以及所谓全固态存储系统。

如果说在这9年里的前4年，企业级存储算是慢慢悠悠地自我欣赏式发展，那么后5年基本是在小步快跑了。一下子爆发的众多变化，都发生在后5年里，企业级存储显得应接不暇、不知所措。云计算、大数据、固态存储、开源、软件定义等各方围剿，使得商用企业存储好像找不到出路，各个厂商绞尽脑汁规划下一代产品到底应该是个什么样子，以及整体战略需要怎么调整。

就在两年前，国内某存储厂商为应对海量低成本存储场景，设计了一套与2005年国外那个厂商类似的方案，也就是在每块磁盘驱动器上前置一个Dongle，基于ARM处理器，这个Dongle相当于一个Mini Storage Controller，功能方面，其在硬件层后端通过集成的SAS/SATA控制器访问并管理这块盘，前端则通过以太网口来传输封装之后的访问协议（比如Object对象访问协议），核心软件层是一个精简的Linux内核，包含SAS/SATA Host驱动、以太网设备驱动、块设备驱动、卷管理层、对象/文件管理层、对象访问协议、TCP/IP协议层以及管理监控Agent等。也就是说，将一块传统的以Block形式访问的磁盘通过加一个转接板，变为了一块以对象Object形式访问的磁盘，如果向其软件层加入更多协议，那么还可以变为iSCSI Target、NFS/CIFS Export，当然实际上一切都受限于ARM的性能。每块对象盘连接到以太网上，再通过一个或者多个冗余/AA的总控服务器来管理这些磁盘，并通过这个总控服务器集群向外提供空间和服务。

IP硬盘——玩玩？

也就在最近，希捷与这家存储厂商联合推出了被命名为Kinetic的硬盘，宣称其直接提供对象访问接口，并向应用提供API以进行数据访问和监控管理。这个产品相当于把之前的转接板去掉，把核心软件直接运行在硬盘背面的控制芯片里。处理芯片的一般架构是一个或者多个ARM/MIPS core与一堆外围电路（比如XOR、ECC/CRC、加密、压缩、PHY等）组成，而ARM/MIPS Core平时不参与数据的传输，只是控制数据的传输，否则会由于过多的内存拷贝而性能根本达不到要求，所以一般来讲一款处理芯片中的通用CPU模块，绝大多数时间负载并不高，这也就为在处理芯片中集成更多的软件功能提供了技术空间。

但是别指望这种低功耗CPU能胜任事务级在线处理，跑跑一般的数据收发、简单的协议处理还是可以的；也别指望其能胜任高IOPS的场景，每一个IO处理耗费的CPU资源是不容小觑的，包括中断、协议处理、内存拷贝等在内的流程对CPU耗费很大。但是低负载、以带宽吞吐量大块连续IO为主的场景下，这类处理器能够很好地胜任，尤其是在只带一块磁盘的情况下，那就更是小菜一碟了。所以这种产品的基因决定了它的应用场景，也就是比如冷数据存储场景或者备份等海量低成本存储场景。

综上，我们暂且简称这种硬盘为“IP硬盘”或者“对象硬盘”。其与传统的存储架构本质区别在于，传统存储控制器属于集中式控制器，用一台或者多台集中式的高性能控制器，通过SAS/SATA适配器接入数量有限的磁盘，最小的比如Raid卡，比如Adaptec by PMC最新的产品可直连24盘或者通过扩展柜连接256盘，最大的比如高端商用企业存储，可以管理多达3000多块盘，前端终结了SAS协议，转为使用FC或者iSCSI、

NAS或者对象等协议，通过集中的、单一的访问点来访问所有磁盘经过虚拟之后的空间。而Kinetic的架构则属于分布式微型控制器，有多少磁盘就有多少个访问点。

说到这里我们就要仔细地去分析一下，这种新架构带来的优点和挑战在哪里。毋庸置疑，其优点是支持大规模并行访问，因为访问点是分布式的，有多少磁盘就有多少访问点，那么应用或者客户端程序可以直接并行地访问所有连接到以太网上的磁盘，体系效率较高。当然，其代价就是访问节点的管理上，需要被软件定义。对于集中式的磁盘控制器，对磁盘的管理，比如监控、容错、性能优化、空间管理等，都由集中控制负责，上层不需要关心，而新架构下，直接暴露了底层的磁盘，那么这些逻辑就需要被挪到上层软件层中去执行，也就是所谓软件定义，那就需要用户具有一定的技术开发能力去驾驭这个新架构，或者由厂商做这一层的开发，但是相对于在外部设备里开发这一层来讲，在用户的OS里做这个管理层，其主要难度在于兼容性，用户的OS千变万化、环境千变万化，兼容性很难保证。所以这类产品应用到互联网后端的可能性较大，一般企业会吃不消对其日常维护的开销。

那么再看一下互联网企业，假设如果依然利用现有架构，比如1U通用服务器，加一个SAS/SATA Raid/HBA适配器，接入12/16/24盘，然后在服务器上进行空间管理、协议转换，底层Raid控制器实现数据的小范围冗余容错及性能优化，在所有服务器上运行分布式文件系统来执行数据的大范围容错和均衡，这样做的好处是对上层来讲复杂度降低。同样是1U服务器，如果访问Kinetic架构，SAS/SATA Raid/HBA就不需要了，直接通过以太网，那么原本由SAS Raid卡做的工作，就需要用软件去做，需要用户自己或者厂商开发一层逻辑，而且这层逻辑要么是分布式部署的，要么是非对称集中式部署在一个带外控制管理节点上的，这

种做法基本上就是将磁盘进行非对称的外集群化，供上层的服务器集群访问。如果使用1GB以太网连接每块磁盘，其带宽相对目前主流的6GB SAS/SATA来讲会降低，时延也会增加。

结论，不管是对于互联网企业还是传统企业，一个集中控制设备或者软件层都是需要的。Kinetic架构的优势在于，降低了访问粒度，提升了大范围内的访问并行度；其劣势在于，性能域扩大，管理域也随之扩大，故障域也随之增大，传统1GB以太网带宽和响应速度有限，对于冷数据这类场景，传统架构在性能、成本、管理上是否已经真的无法满足需求？是否有必要去这样折腾，还是个需要考虑的问题。

附录3 新技术将如何影响数据中心存储系统

自诩为一个可以见证中国存储发展的存储界老混子，不得不承认，存储的发展真的是太快了，以至于很多技术还没大展宏图，就发现眼前的这片森林已经今非昔比。我想这也是当前很多存储厂商、集成商所面临的困惑之一。

有很多人曾经和我讨论过诸如“我们下一步到底该做什么”的话题。有些厂商做法很简单，一线品牌厂商做什么，就跟着做什么，这样最保险，但是没有一定实力的厂商也玩不起。对于二三线厂商，事态尤为严重。换在几年前，我想很多厂商目标都比较明确。但是，近几年新技术和新概念爆发式的产生，而存储领域的产品集成开发周期又相对较长，这是导致目前众多厂商迷茫的原因之一。等你的产品出来了，却发现走错了路，或者窗口期已经过去。

本文试图对当前多个存储子层里的多项技术做简要分析来获知它们对传统体系的影响。

1. 存储介质——闪存和**SMR**磁盘

机械磁盘作为在线主存储介质的角色，可以说几十年来没变过，是各种存储技术里最稳定的一个了。然而闪存的出现，将要改变的不仅是存储介质，更将会改变整个存储生态链。

闪存作为新一代存储介质相对于机械磁盘的优点不必多说，一个更加值得思考的问题是，由于闪存并不像机械盘一样需要高精尖的技术，其入门门槛较低，尤其是闪存控制器的设计生产，目前可以说是遍地开花，国内已经有多家自主产权的闪存控制器及外围产品。

希捷和西数这两家机械磁盘的巨头如今也开始居安思危了，一方面积极研究和融入闪存技术，另一方面也在积极朝着产业链下游发展，从希捷收购Xyratec的动作就能略感一二。在未发生颠覆性变革之前，产业链上游是既安全又稳妥的地方，但是当即将发生颠覆性变革的时候，整个产业链都会受到影响，此时便会出现群魔乱舞的壮观景象。比如与云计算看似无关的希捷，联合下游厂商推出适用于云计算大数据的新型磁盘Kinetic，同样，Intel也在积极参与云计算大数据领域。包括一向低调的PMC-Sierra也在收购了板卡厂商Adaptec之后又收购了闪存和PCIE控制器厂商IDT并开始厚积薄发。

闪存对传统的下游存储厂商的影响也将会是巨大的。

首先，基于机械磁盘介质所积累的成熟的传统架构基础面临崩塌，包括硬件设计（比如尺寸、散热、承重和空间布局等），也包括软件设计（比如数据布局、IO性能优化和故障预测及恢复等）。这也是为何仅仅把传统存储系统中的机械盘替换为SSD之后却发现根本无法发挥出

SSD性能的原因（当然制约SSD性能发挥的本质因素还有一个，会在下一节讨论[DW5]）。

其次，传统存储高大上的形象也会被闪存彻底摧毁，原本松耦合的各种大部件搭配起来的“巨型机器人”将变得非常小巧。最极端的情况甚至可以直接用一块PCIE Flash卡替代，连影子都消失在了服务器机箱外面。磁盘存储将退居二线，成为真正的备份用二线存储，使用步骤或许会是“开机→备份→关机”。磁盘存储将会成为下一代人眼里彻底的淘汰产品，就像卡带机一样。

再次，拖累传统磁盘存储的另一个包袱，就是那些华而不实的软件功能，包括自动精简配置、重删、快照、分层/缓存和复制等。这些软件功能除了其中几个较为常用之外，其他可以说是鸡肋，但为了商务竞争又不能没有，陷入恶性循环。上述软件功能中的每一项，毫不客气的说，都是影响性能的。分层和缓存实际上是增加了相对性能，而降低了绝对性能。除了快照、分层和远程复制之外，其他软件功能多数时候都不为人所用。

用户似乎越来越追求傻快的存储。这一点在面对“软件定义”概念时更有说服力了，硬的更硬，软的更软，这更进一步拉低了存储系统的门槛。抛开了这些包袱，利用闪存，越来越多的全闪存存储厂商出现了，而这些全闪存存储为何基本都没有出自传统一线存储厂商[DW6]，原因显而易见了。

磁存储领域的一项新技术是SMR（叠瓦式磁记录）技术，这项技术提升了存储密度，却不能保证随机写的性能，这一点从原理上讲更类似于固态介质的Page与Block之间的尴尬。希捷等磁存储厂商也正在研究是否要在SMR磁盘内实现类似Flash的管理方式。SMR磁盘面向一写多

读场景，适用于大数据、备份等特定领域。下一步还有热辅助磁记录技术，但是迟迟未能商用。看来磁存储在性能瓶颈之后，可商用的容量瓶颈也即将到达。磁存储淡出舞台是大势所趋。

2. 底层框架——芯片、底软和通道

闪存的出现，会影响生态链上的所有事物，这其中也包括了最底层的芯片、底层软件和数据通道。

芯片

芯片要有足够强的处理能力来承载起闪存强悍的IOPS性能，包括Flash控制芯片、外围协议控制芯片（SAS、FC、以太网）以及主机CPU。

芯片的提速手段有三种，第一是提高内部数据带宽，增加通道数量和带宽，第二是提升器件频率，第三是将各个子器件进行拆分，增加并行度，在相同电路周期内可并发执行更多的指令。然而，没有免费的午餐，上述任何一个动作，要么会增加芯片的功耗，要么会增加面积，这些都是弊端。

目前一线厂商PMC的主流存储控制芯片实际功耗都控制在15W上下，即便是最新的SAS 12GB主控芯片，由于制造工艺的提升，功耗反而比6GB产品有所降低。到目前为止，主流存储芯片都是基于MIPS核心+外围加速电路，MIPS是被公认的RISC通用处理器领域最经典的代表，然而ARM的猛攻也渗透到了存储芯片领域，在低端市场占据了席位，包括4端口SATA控制器、低端SoC等等，ARM和MIPS也会在存储芯片领域持久对峙下去。Intel则由于功耗问题，颇有绑死x86平台走到底的趋势，移动终端失策，卖掉电视部门，靠Atom在大型数据中心领域与ARM抗衡，不知道格局能维持多久。

底层软件

底层软件方面也是制约存储性能提升的一大屏障。拿Linux为例，Block层、SCSI中间层这两大制约IO性能发挥的重量级软件层，在机械盘时代发挥了重要作用，然而在闪存时代，其变成了严重拖累性能的罪魁祸首。繁冗的扫描机制，低效的互斥队列和捉襟见肘的队列数量，陈旧不堪的SCSI协议，这些对闪存来讲都是头疼的事情，目前闪存产品不得不选择越过SCSI层而直接注册到Block层，然而却丢失了SCSI层提供的兼容性优势，导致各家在Block下层的协议实现不统一，增加了开发成本和管理开销。而业界关于这方面的两大协议阵营——NVMe以及SCSIe也在飘忽不定，导致厂商不得不两手准备，痛苦不堪。目前似乎NVMe更胜一筹，这也是存储底层部件巨头PMC-Sierra收购IDT的原因之一，IDT的Flash控制器是业界第一家完整支持NVMe的成品控制器，目前已经在多家固态存储产品中被应用。

数据通道

协议接口方面，基于SCSI体系衍生而来的势力有三股，一个是FC，另一个是SAS，还有一个是iSCSI纯软件方案。然而NVMe与SCSI从头到脚都是两套完全不同的协议，虽然它们的目的是一样的，就是把数据从介质里传到内存里，但是NVMe除了相对SCSI协议做了很多精简之外，还在数据结构方面做了很多优化，充分发挥底层介质的并发访问性能。Linux开源社区最近也在研究如何优化SCSI层的问题，看来SCSI是去是留已经是个问题了。FC通道前端目前正在逐渐被万兆以太网蚕食，而后端则在几年前就已经被SAS全盘端掉。

SAS之所以没有端掉FC前端有两个原因：其一是因为FC前端体系

并非只存在于存储设备内部，而还涉及到交换机，其存量市场并不是仅仅通过替代掉存储设备的前端通道卡就可以占领的；其二，SAS在光传输方面略显迟钝，究其原因在于SAS光协议对于链路协商方面的一项技术实现争议了良久，直到很晚才确定。

FC也必将淡出舞台。然而，其接替者并非只有以太网或者SAS。还有另外一项更为前瞻的通道技术，那就是PCIE。目前我们所熟知的以太网、FC、SAS、Infiniband等通道协议，在主机层面无一不通过PCIE接入系统IO总线。之前的“远距离”传输概念，正在变得模糊，多“远”算是“远”，如果PCIE能够“远”到一定距离，还要以太网作甚？这个问题问得好，但是PCIE并不是万能的，PCIE目前缺失很多交换网络特性，毕竟之前一直是在系统总线领域，出了总线，就得长距离交换和路由，这方面就得靠以太网和TCP/IP了。然而，同样的理论，在目前和将来的数据中心领域可不见得能套用。目前的数据中心有苗头正在朝着紧耦合方向发展，也就是之前一个机架内的服务器之间是松耦合的，现在要变得以机架为单位，机架内部紧耦合，机架外部松耦合，此时PCIE就有用武之地了，机架内部完全基于PCIE矩阵。针对这个前沿方向，目前Intel以及PMC-Sierra都有研究并且有了DEMO。当然，对SAS和SATA的兼容一定是要考虑的，SFF8639接口标准其实是一个三模式（Tri-mode）的接口，把SATA、SAS和PCIE打包到一起，后端则根据前端接入设备类型路由到SAS Expander/Controller或者PCIE Switch上去。目前看来这个接口已成定局。

3. 数据结构——RAID 2.0、ErasureCode、分布式及开源

RAID 2.0

硬件平台之上的软件，也在风起云涌地变化着。传统存储领域可炒作的概念已经没有了，然而创新又迟迟未见。RAID 2.0被几家厂商在炒作，但终归也是RaidEE技术的升级版。另外，RAID 2.0与现在多数技术一样，只是提升了相对性能，而没有提升绝对性能，也就是当磁盘达到一定数量的时候，这项技术才会显示出优势，但是依然赶不上相等数量的磁盘在传统模式下的绝对性能。RAID 2.0对数据的处理，已经不亚于一个文件系统了，过多的数据碎片影响了绝对性能，但是大量的磁盘堆砌又可以掩盖这一事实。其所获得的唯一一个绝对好处是重构时间的大幅降低，然而却牺牲了平时的绝对性能。

ErasureCode

ErasureCode技术也不是什么新鲜事。RAID 6以及RAID DP技术很早就出现了，那时候人们已经发明了可以容忍更多磁盘同时损坏的技术，只不过受限于随机写性能而没有将其商用。但是时过境迁，大数据时代读多写少，再加上数据量大，RAID DP（Double Parity）、RAID TP（Triple Parity）甚至允许更多磁盘同时损坏的算法，就又冒出头来了。

分布式

Scale-Out是传统存储领域对“分布式”的一个包装词，然而传统存储

理解的分布式和互联网及开源领域所认识的分布式骨子里还是不同的。传统存储厂商的分布式不是廉价的分布式，它们的分布式完全是为了解决Scale-Up模式的天花板；而互联网和开源的分布式骨子里为的就是廉价。表现形态也不同，前者虽然实质上也是x86服务器+分布式软件管理层，但是依然略显高大上。

开源

开源的风潮体现在最近的一个新概念里，那就是所谓“软件定义”了。软件定义让二三线厂商出师有名，直接挑战传统一线大厂的权威地位，这一点从近期一些二三线厂商直截了当的露骨演讲即可知道，矛头直指一线垄断大厂，似乎在当头棒喝“凭什么你们就是高大上”。

4. 用户体验——接口、访问方式及展现

在对存储的访问接口方面，新的访问接口近几年在互联网的带动下也发生了爆发式增长。传统领域一直在鼓吹所谓“统一存储”，鼓吹了近十多年，早就炒烂了。对象、key-value、文件、块是目前来讲主流的4种访问形式，其中文件又包含多种子类型（比如NFS、CIFS以及各种分布式文件系统访问协议），块又分为FC、SAS、iSCSI。不管访问形式如何，它们本质都是一样的，都是对一串字节的请求和回复，只不过这串字节在不同应用场景下的归类不同罢了。

在用户体验方面，传统存储做得较差。但是随着互联网风潮来袭，重视用户体验、应用感知、QoS等更加接近用户层面的功能越来越受到重视。

提示： 笔者之前所设计的存储软件套件SmartX Insight就是从用户体验方面来入手，增强存储系统在整个系统内的“存在感”，改变传统存储一副道貌岸然的样子。我想这样更有利于黏住用户，从而扩大及延长存储系统的生存空间和时间。

5. 闪存与数据中心——SATA/PCIE及应用场景

目前来看，数据中心对Flash的渴求主要集中在几个固定的应用场景，前端来讲，比如CDN，ISP的带宽是非常贵的，必须充分利用，所以硬盘必须不是瓶颈。后端则是各级缓存场景，包括各类分布式数据库系统、分布式文件系统的前端基本上都是放了一级或者两级甚至更多级的缓存，RAM毕竟还是很贵的，而且容量有限，主要用于第一级缓存直接应对前端的压力，Flash则可趁机占领一部分后置缓存空间。

数据中心对SATA接口SSD的应用占据了总体形态的大概90%，剩下的10%主要是PCIE接口的Flash，前者基本被Intel独占，后者则是花开几朵，其中也不乏国内厂商。

PCIE Flash是大势所趋，尤其是支持NVMe标准的设备。但是目前的形态却不被看好，别看当下多人在此领域角逐。当前形态存在的问题是维护困难、版型太大，这些均不符合数据中心对硬件资源的要求（一个是维护方便，另一个是资源性能和容量粒度要尽可能低，以便于灵活拼搭）。而基于SFF8639接口标准的设备相信马上就会遍地开花。

综上所述，各种新技术对存储系统的方方面面产生了很大影响，如今IT界概念频出，五彩缤纷，众多的存储厂商唯有分析历史，分析当前，才能看清未来。

后记

石灰吟

千锤万凿出深山，
烈火焚烧若等闲。
粉身碎骨浑不怕，
要留清白在人间！

——于谦

各位朋友，非常感谢您能看完此书。如果您对这本书有何建议和意见。可以发送邮件到122567712@qq.com，我当万分感谢！

另外，还可以到本人博客留言或者邮件讨论本书相关的内容。

最后，实在想不到拿什么送给各位以表谢意，就送各位一首诗，也送给我自己。

书湖掌中

闭关数载修此书，
练得秘籍献江湖。
七星降龙独孤掌，
多少豪杰醉其中！

书中角色最后归宿：

七星大侠：开天鼻祖，光芒永照。

张真人：百年求道，一生孜孜不倦、德高望重、悬壶济世、鞠躬尽瘁、死而后已。

微软老道：承蒙张真人赏识，不负众望，成为武林盟主。

无忌：革命之后，到处求仙访道，不知其踪。

老T：把持武林交通系统，依然向最后一块阵地不断进攻。

FC大侠：把持着那最后一片领土，与老T对峙到底。

冬瓜头